High Level Synthesis of Memory Architectures

Hamish Fallside

A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy
to the
University of Edinburgh
1995

Abstract

The development of high level tools for electronic design has been driven by the increasing demands of an ever more complex design process. The diversification in the use of electronic circuitry requires design tools tailored to application specific domains. Intelligent synthesis requires domain specific knowledge in addition to general synthesis techniques. The preponderance of synthesis systems in domains such as Digital Signal Processing is indicative of this need.

Methods are presented here for the synthesis of memory architectures in one such domain: image processing. The research concentrates on performance synthesis. The techniques presented aim to optimise the design so as to minimise the memory access bottleneck of the eventual hardware implementation.

The development of a synthesis system is described which served to support the research. Algorithmic descriptions, coded in C, are processed by the tool in order to produce a structural description of a memory architecture able to implement the presented algorithms in hardware. Data flow and dependence analysis techniques are employed, these address the "high levelness" of the input algorithm, an important task if the designer is to be relieved of low level design detail.

Methods for organising the algorithm's data in, and it's access from memory are presented, and experimental results are included. The organisation of data in memory is accomplished as part of the scheduling process for the user algorithm. The methods aim to optimise the hardware implementation by maximising the utilisation of the memory resources allocated during synthesis.

In dealing with the access of data from memory, methods are presented for the automatic detection of memory inefficient structures in the user description, and their transformation into a representation yielding synthesised designs with greater memory throughput. Such designs are better able to support the user's algorithms within desired performance limitations. Examples are included which provide an evaluation of the techniques' efficacy.

Acknowledgements

My thanks to my supervisor, Professor Peter Denyer, for his ever-open door and the constant encouragement he has given me during my time at Edinburgh University's Department of Electrical Engineering. I'd also like to warmly acknowledge the help and guidance of my second supervisor, Dr David Renshaw, during the writing of this thesis.

I'd like to thank VLSI Vision Ltd for examples, food for thought and valuable experience over the years. Also to the various members of the Integrated Systems Group who provided me with examples, particularly for those that appear here from Henry Bruce, Ann Duncan and Colin Ramsay, thankyou.

Mags Chalcraft and Maureen Fallside gave invaluable help by proof reading, editing and helping me think beyond the words to the language, for which I take full responsibility. Lastly, enduring thanks to the many friends for the encouragement and understanding they have given me whilst I have been completing this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

By the turn of the century Moore's Law[1] tells us that integrated circuits will contain as many as 96 million discrete components [1]. Making effective use of such technology with today's design tools will not be practical, especially in a rapidly expanding market-place that demands ever shorter lead times between design conception and implementation. Furthermore, the increasing diversification in the application of electronics requires that greater numbers of designers be able to harness the technology for their own ends. This means that the learning curves associated with Electronic Design Automation have to flatten so as to allow less specialised designers access to the technology.

In order for design engineers to harness such technology effectively, the tools that they use will have to be orders of magnitude more powerful than at present. Such tools will have to provide high levels of design abstraction so that users are separated from the complexity of such large circuits at gate and transistor level. In addition the designer must be able to explore effectively multiple solutions prior to fabrication of the device, so that the appropriate solution can be found. This requires that accurate performance data is obtainable throughout the design process to allow rapid prototyping for test.

In [2] J Allen refers to the evolution of the relationship between design synthesis for VLSI and design performance testing. He describes three overlapping epochs since the introduction of the first integrated circuits in the late 1950s.

[1] Moore's Law states that the density of transistors on a silicon chip doubles every 18 months

The methods developed during these periods have been a response to the effects of Moore's Law and have involved a dynamic feedback relationship with the changing market needs of the industry.

During the first of these epochs, designers initially had to rely on results from fabricated circuits in order to evaluate designs' performance. As CAD tools developed it became possible to extract net-list information from layout and use this to simulate the circuit behaviour. Design synthesis was laborious and required great attention to low level detail for successful results.

The development of macro-generators and parameterisable cells heralded the second of Allen's epochs, around the start of the 1980s. Such tools gave designers access to timing and performance data before any polygons were laid out. This allowed design decisions to be experimented with earlier in the design process, leading to shorter synthesis/test cycles.

Full integration of such tools and techniques has arguably not fully occurred even now as the CAD community "strives to enter" Allen's third epoch of "Intelligent Compilation". The stated aims of this new era are to combine optimisation strategies and high level design specification. This allows engineers to refine their exploration of the design space by tuning designs at the algorithmic level whilst quickly obtaining performance feedback through characterisation and simulation obtained from the same design specifications.

Only with such tools can the integrated circuits of the next century be designed. Much of the work performed by IC designers at present will have to become automated in order to increase their productivity. This is commensurate with the changing market needs which have been developing in the micro-electronics sector over for past decade or more. The research presented in this thesis is an investigation into tools that might meet the demands of this changing technology.

1.1 The Need for High Level Synthesis

Without computer tools for simulation, synthesis, visualisation and organisation VLSI design would be a formidable task, which would require many person years of to complete each IC. As design complexity increases, driven by the unremitting Moore's Law, the level of design abstraction of such tools has to increase at the

same time. This keeps the user separate from the low level detail, ensuring they do not become engulfed by the design.

The beginnings of Allen's third epoch, that is the development of intelligent compilers, began in earnest during the the 1980's. Some efforts had been made previously, for example IBM's ALERT system [3], and the Design Automation project at Carnegie Mellon University [4]. These, however, were largely academic projects. The electronics industry during the 70's was concentrating more on layout and logic synthesis tools.

Over the decade and into the 90's, a number of different and diverse communities contributed to this research effort. For example Kowalski's artificial intelligence perspective [5] and the software compilation techniques used by Tricky [6]. The techniques from parallel processing theory applied to loop pipelining [7]. The development of novel languages [8], from within the CAD community itself and, in the extreme, evolutionists [9].

This diversity of input, coupled with the continual growth of the application domains addressed by high level synthesis, has led to compilers for synthesis that are highly specialised. This focussing means that "Everyman's Compiler" is not a reality and perhaps never will be, at least not at the abstraction levels we deal with at present. One might envisage, on the other hand, a knowledge-based tool to aid the designer in deciding which compiler to use!

1.2 Focused Synthesis

The individual needs of different fields of application for high level synthesis are not necessarily portable between different areas. The demands of producing designs of a quality comparable with the best human designs are incompatible with a single tool philosophy. The higher the tool in the synthesis abstraction level the truer this becomes.

For example, the techniques employed to synthesise an asynchronous microprocessor from an instruction set specification, as employed in [10], will not be the same as those required to synthesise a dedicated data path from an algorithmic description as in SPAID [11]. There will be some overlap between the two sets of techniques used; but not enough that the two might reside in the same

tool. These differences are largely due to the different application domains being addressed.

Thus in order to keep tools manageable, target application areas, each with their own knowledge sets, are separated within the synthesis domain. Tools to address each domain are developed, employing common themes and also utilising specialised knowledge about the target domain.

In the future one can imagine synthesis systems that guide the user towards particular tools, which allow them to try a range of architectural styles for their particular design. Knowledge Based Advisory Systems will suggest the best avenues of approach based on factors such as system power consumption, design time, compatibility and connectability with other systems. This will be in addition to the more contemporary synthesis considerations such as power, area, speed and cost.

The development of system level synthesis, as for example in APARTY [12] and the System Architect's Workbench [13] is a move towards this idea of "meta" synthesis. The interest in systems integration and synthesis frameworks provides further evidence of this trend. This would appear to fit well with the contemporary idea of "open" systems.

However, at present, high level synthesis concentrates on focused application domains. These segment broadly into general ASIC design, DSP ASIC design, Video and Image Processing ASIC design, microprocessor and general processor applications, parallel processing systems and, increasingly, FPGA design.

Areas of overlap between these fields are addressed separately by some compilers, e.g. pipelining [14] and retiming [15], as well as synthesis specific problems like scheduling and allocation [16]. Other topics are more exclusive to certain areas than others. DSP synthesis generally requires highly multiplexed systems for dealing with high frequency sample rates [17]. The field of synthesis for image processing builds hardware for processing large quantities of data at medium to high frequencies, here the memory design is important [18].

The importance of these two areas is under-pinned by the increasing use of information technology in modern society. Digital communications, multi-media, the increased use of computers for monitoring in manufacturing and the general environment all stoke a demand for bespoke designs in smaller quantities than

has been traditional for chip fabrication.

1.3 Memory Synthesis

This thesis approaches one such focused topic: the synthesis of memory architectures for image processing applications. The development of on-chip cameras [19] in conjunction with Moore's Law has given rise to increased integration of sensors with processing electronics [20]. In the future this, and other areas will further fuel the demand for synthesis tools which aid the designer in meeting smaller turn around times for ever larger and more complex designs.

The area of image processing is typified by the iterative processing of large amounts of data at high sample rates. Actual processing of the data is often relatively simple, but the sheer quantity of data handled in each cycle is usually large. For example in video processing, video images of up to 1 Mbyte might be processed at a sample frequency of 25 Hz, which gives 0.04 μs to process each byte in a serial manner. This would allow for a very small number of accesses to the data, of the order of three or four. If each raw data byte has to be addressed a number of times, then a significant amount of the time available for processing each datum can be spent accessing memory.

Employing faster memories increases the rate at which data can be accessed, but there is no escaping the absolute limit imposed by the processing algorithm itself, which will demand a certain order of access particular to the task being performed. So, in order to build hardware capable of implementing such algorithms, attention must be paid to the I/O requirements of the algorithm. For an expert designer such an implementation would necessarily involve the construction of a memory hierarchy capable of dealing with the high data throughput, as for example in [20].

This is well and good for the expert designer. However, the rate of growth of the electronics market dictates that not all design can be done by such experts if such expansion is to continue. Moreover, the advent of new technologies, particularly programmable logic [21], and their greater market penetration means that more and more people will be doing electronic design in the future. In order to satisfy this demand, tools with a higher level of sophistication than those

traditionally used up until now will be required.

Such tools need an input format that is able to represent algorithms at a high level and which does not require specialist design knowledge embedded in the description. Synthesis tools should take the synthesis out of the hands of the designer, allowing them to concentrate on developing the applications and leaving the design of the hardware to the tools themselves.

The work presented here utilises input expressed in the C programming language for representing the algorithm. The choice of programming language is arbitrary, C was chosen by virtue of it's popularity as a general purpose language and because of the availability of public domain grammars and other compiler tools.

In order to perform optimisation of the hardware produced, two approaches have been taken, exploiting potential access parallelism inherent in the algorithm and the application of high level transforms to the original description in order to improve the implementation.

1.4 Architectural Optimisation

Firstly, synthesised architectures are optimal for the (memory) resources available to them. This is achieved by fully exploiting the potential parallelism of memory accesses in the code description. Parallelism can only be exploited when there are multiple resources available at any given time. Scheduling, therefore, is performed for each different architecture produced.

In order to find the parallelism inherent in the memory access profile, the communication between memory and core has to be analysed. This is accomplished via the use of array dependence analysis, which provides information concerning the precedence of array statements with respect to one another.

This analysis also provides data useful in transforming the input description, so as to improve subsequent synthesis stages of the algorithm. These transforms supply the knowledge that would be provided by the expert designer concerning the architecture of a memory hierarchy.

The use of high level transforms in synthesis is increasingly common. Software compiler optimisations have been employed in a number of synthesis systems, for

example [22], [4], [13], [23], [24]. These transforms provide a means of improving synthesized hardware by manipulating the input description so that it gives a better basis for synthesis, whilst retaining the original semantic meaning given by the user.

In some respects such transforms accomplish little more than a "tidying" or normalisation of the written description. If the user spends more time polishing and hand-optimising the code, the transforms become redundant. This is the principal argument against high level synthesis as a whole: if users spent more time over particular design details there would be no need for high level specification or high level synthesis.

Adherents to this view must necessarily tolerate long lead-times for design and a high level of design knowledge on the part of the user. Moreover, design activity would be confined to a small, highly-skilled group of people, but working in a continually expanding market place.

In addition the burden of verification for increasingly complex designs still remains largely in the hands of the designer. The very nature of low level design makes it a highly error prone activity. Designers have to cope with multi-levelled abstractions and the complexity of manipulating such organisations requires an awareness of detail that is often beyond the capabilities of a single person. The practice of designing in teams increases the opportunity for errors to be introduced. This problem becomes only greater with the inexorable increase in integration.

In order to meet the technological demands of the next century, users will have to be freed from such design detail. Languages and methods of specification of hardware are essential. Such tools will allow them to concentrate on the development of the algorithms to be implemented, leaving the more onerous tasks of low-level design and design verification to the tools. High level synthesis will have to provide such tools.

1.5 Thesis Overview

Below is a short summary of the following chapters, giving an overall picture of the thesis.

- Chapter 2

  This gives a review of relevant work in a number of different fields. It covers general high level synthesis, with particular emphasis on scheduling and allocation methods, transforms employed by the various synthesis systems and issues relating to input representation.

  Other synthesis systems covered are those that pay particular attention to memory architectures. Amongst the few silicon compilers that directly address this topic are the CATHEDRAL and PHIDEO tools. These two systems are covered in detail.

  Another common theme examined in this thesis and shared by some other systems is the synthesis from high level programming languages. Some issues relating to this topic are examined in the literature review and are also covered in Chapter 3.

  An important part of the thesis work is the use of high level transforms during the synthesis process. This is a recognised feature of high level synthesis which allows the optimisation of designs through the exploration of the solution space. In addition to the transforms already generally accepted some new techniques from the field of super-compilers is presented.

  Many of these transforms result from the analysis of dependence information between array statements in software code. A review of the methods utilised in this analysis is given, serving as an introduction to their use in later parts of this work.

- Chapter 3

  This chapter introduces a compiler tool that acts as a platform for the techniques presented here. The tool is called CSiC , for C to Silicon Compiler, and generates memory hardware solutions for algorithms written in the C programming language. These solutions are optimal in terms of the data organisation in memory and the access of the data.

  An introduction to the tool is given, with an overview of it's goals. Some exploration of possible alternative strategies is made and the reasons for developing CSiC are presented.

There then follows some discussion of issues relating to the input given to CSiC , the limitations imposed on the language and the format expected.

The remainder of the chapter introduces the preliminary phases of the tool's operation. Parsing and preprocessing are covered, followed by the basic analysis which provides much of the information required by the later stages of synthesis.

- Chapter 4

  The generation of memory architectures by CSiC is examined here. This covers a strategy for defining the behaviour of the synthesized hardware through the scheduling of memory accesses. This is analogous to the scheduling of logical and arithmetic operations onto a data path in traditional synthesis.

  Scheduling takes place at both the local and the global level, these are both covered here. The related topic of the allocation of data to physical memory is covered as well. These two techniques are closely related, as are scheduling and allocation in the usual sense.

  CSiC generates a range of architectures, exploiting opportunities in the input description for parallelism at both the local and global levels. In order to select optimal solutions from these some selection has to be performed. The generation of timing profiles for the completed architectures is presented alongside a costing strategy.

- Chapter 5

  In order to improve the architectures synthesised, CSiC applies high level transformations to the original input description. This process and the transforms that are involved are covered in this chapter. The transforms originate in the area of software compilation for vectorisation and are previously unapplied in the field of high level synthesis.

  The transforms operate at the source level, this provides a useful means to verify their function as the transformed internal description can be emitted as C code and recompiled by the user.

The aim of the transforms is to optimise the code for memory accessing by manipulating the sequence of accessing that occurs in the original algorithm, whilst keeping the semantic meaning intact.

- Chapter 6

  Some results from the tool are given in this chapter, from the compilation of sample applications in the image processing domain.

  Two applications have been chosen, both from the domain of image processing. The first application is a method for image compression [25], using Vector Quantisation [26]. The second example is part of fingerprint recognition system developed in Edinburgh [27], and implemented in hardware as an ASIC design [20].

- Chapter 7

  The work presented is summarised in this final chapter and conclusions drawn. Various lessons learned as a result of this research are expanded upon and suggestions for future development of the tool, and the field of memory synthesis are included.

Chapter 2

Literature Review

This chapter presents a review of areas relevant to the compilation of memory hardware systems from algorithmic descriptions. It is necessarily broad in its scope, covering the fields of high level synthesis [28], and techniques used therein: synthesis from high level languages, high level transforms, scheduling and allocation. Also reviewed is relevant work from the field of software compilation, and some results from analysis techniques that have important results in the field of memory synthesis.

There has always been a close association between behavioural level hardware synthesis and software compilation, as for example in [6]. This promises to continue, especially with the advent of new technologies, such as FPGA technology [21], which portend a further lessening of the distinction between software and hardware [29].

Additionally, for synthesis to provide the tools for fast lead time product development the issue of system synthesis has to be addressed, that is the design of entire sub-circuits of chips and components. Behavioural synthesis will provide a method of achieving this rapidly and effectively. However before this can occur new techniques have to be developed for the partitioning of behavioural descriptions and for the design of those partitions. This chapter aims to cover what has been done in this field so far, and attempts to introduce relevant work from other fields.

2.1 High Level Synthesis

The field of high level synthesis is a broad one, perhaps more so with the realisation that, in order to achieve results comparable with human designs high level synthesis tools must be tailored to meet the demands of the application specific area. The idea of an all purpose silicon compiler is unrealistic in this context as no one tool will ever hold all the required design knowledge.

This is due to the wide range of applications for which hardware systems are now required. The spread of communications technology, the industrial and commercial interest in vision and image processing, and the general expansion of digital hardware have all contributed to the increasing need for better design support. This support must come in the form of tools for design.

The importance of high level synthesis cannot be overstated in this context: the specialist expertise required for the design of digital circuits is a limited resource in today's industry. For expanding market needs to be met effectively this specialist knowledge has to be made available to a growing base of would-be designers, and at a reduced cost. High level synthesis has the potential to achieve this goal.

The synthesis task is the transformation of behavioural descriptions into structural descriptions. These can then be implemented physically, as digital circuits. The behaviour expressed in such a description is the set of actions performed on an input that result in an output. The structure that is synthesized is described in terms of components at a particular level of abstraction.

These levels of abstraction are described in the design hierarchy given in [28], part of which is reproduced in Table 2.1. Only the top three levels of the hierarchy are reproduced here, it is these that are relevant to high level synthesis. Similar hierarchies are given elsewhere, the most common of these being the Y chart [30], which gives an abstraction of the different tasks performed in synthesis.

Broadly speaking, high level synthesis is the transformation of behaviours, the first domain column of Table 2.1, into structures, the second domain column. Movement in the hierarchy is downward, although some systems back-propagate results from the lower levels up the hierarchy, in order to improve the synthesis at these levels.

Level	Domains		
	Behaviour	Structure	Physical
System	Processes	Processors Memories	Cabinets Cables
Algorithm	I/O	Processors Memories	Boards
Register Transfer	Register Transfers	ALUs Registers	ICs/Data paths Controllers

Table 2.1. Part of McFarland's Synthesis Hierarchy

The most common type of synthesis performed is at the Register Transfer (RT) level. This implies that some system and algorithmic level synthesis has already been performed. These tasks are usually performed by hand, although more so in the case of system level synthesis for which a comparatively small amount of work has been reported.

One such approach has been to apply transformations to the input description. In [13] methods are described for partitioning designs within chips and between separate chips and boards. This is achieved by applying transformations which create concurrent processes that can execute under separate controllers. The work presented in [13] does not detail the exact effects of the transformations on the eventual designs, but illustrates their effects on the internal representation. This is indicative of the work yet to be done in this field, the value of of many of these transformations are more qualitative than quantitative.

System level partitioning has also been looked at in [12]. Hardware is partitioned using a multi-stage clustering algorithm with the aim of segmenting the hardware under the criteria of control and data transfer and functional redundancy. This method simplifies the design of the different components and their inter-connectivity whilst grouping heavily communicating parts and exploiting parallelism. The authors claim an improvement at all levels of the design process [31].

Also in [32], Simulated Annealing [33] is used for partitioning DSP algorithms amongst separate chips and boards.

The distinction between Algorithmic and RT Level Synthesis is less well

defined. This is largely due to the use of input languages amongst the different systems. Many of these languages[1] derive from programming languages and are capable of expressing descriptions at both levels. This leads to some confusion over what type of synthesis is being performed, often an RT description is used as example input; or, more commonly, the translation from input language to internal format is not covered, and synthesis begins from an RT description.

The CATHEDRAL[2] tool set [34] performs algorithmic synthesis, taking high level descriptions in the Silage language [35] and synthesizes a range of different architectures, depending on the field of application. In particular, the CATHEDRAL II, III and IV tools [36], [37], [38] perform a large range of functions that generate RT descriptions from a Silage input program.

The compilation of Pascal programs into silicon, described in [6], is another instance of an algorithmic synthesis tool. The use of software compiler-like optimisations in this tool to extract a hardware description from the input program is algorithmic in nature.

However, at physical levels below RT there are still choices to be made before a design can be implemented. The choices available at this stage will be determined by decisions made during the higher levels of design. This illustrates one of the problems of high level synthesis, namely identifying in the completed design the effects of decisions made at a high level.

This is especially apparent when constraints such as area cost and timing are considered. Different systems offer different approaches to this problem. In the case of area, one solution is to use standard cell libraries for the RT structural blocks that are used to construct the design. This is employed by the CATHEDRAL compilers [39], the HYPER system [40] and the Sphinx Design System [41] amongst others.

Another approach to the problem, used in the Yorktown Silicon Compiler [42], is to perform synthesis right down to the layout stage and then back-propagate cost information for refining the synthesis at the higher levels.

Alternatively, the selective application of heuristics during synthesis limits the number of possible solutions encountered. Costing is then left until the design

[1] See Section 2.1.1

[2] Section 2.2.1 discusses memory synthesis used in these tools

is complete, when the effects of these techniques can be estimated. All current systems have to rely on such methods up to a point, as there will always be some cost effects that cannot be modelled; and moreover exploring all areas of the design space right down to layout is clearly too great a task.

A novel approach applies a simulated evolution algorithm to synthesis [9]. There are two main components to the system: a GENERATE function which produces a solution to the synthesis problem from a partial solution, using random variations - this mimics hereditary variation: a SELECT function which probabilistically removes high cost components from a given solution, returning a partial solution. The system is seeded with an empty partial solution and the two procedures are iterated until some end position is reached, either that a given number of generations have passed, or a satisfactory solution is obtained. The authors claim the system gives good results compared with other systems, although for longer run-times.

2.1.1 Input Description Languages

The languages used depend upon the level of abstraction at which synthesis starts. System level synthesis requires that separate modules communicate with each other in a structured manner. Ideally this should be inherent in the semantics of the language, to avoid the necessity of explicitly specifying the message-passing mechanism in each script.

In this way the designer is freed from low level detail such as *how* two processes communicate. The compiler may prompt for the type of message passing that is to be used, eg FIFO buffers, token passing, synchronous, asynchronous, etc.

In the Elf [43] system the Ada programming language is used as an HDL for the synthesis of hardware. Communications between different hardware blocks is specified via the rendezvous mechanism for Ada tasks. The system also utilises the language's constructs for concurrency to allow parallelism in the hardware to be represented. As is common in similar synthesis systems a subset of the language is used and concepts such as dynamic memory allocation and recursion avoided.

Similar constructs exist in IBM's V language [44]. This C-type language

supports tasks in a similar manner to Ada, and also implements asynchronous procedure calls and queues. The language was developed specifically for hardware synthesis; this is reflected in it's provision of communication mechanisms for interfacing hardware to the outside world.

Algorithmic level synthesis also has a message passing requirement, although this tends to work more at board level; hence implementation of communications protocols is less standard and more dependent upon where partitions occur and the function of code in each partition. High level procedural programming languages have been used in a number of systems. These are shown in Table 2.2.

Language	System	Ref.
Fortran	HARP	[45]
Pascal	Flamel	[6]
HardwareC[3]	Stanfords Olympus	[46]
C	C-to-Silicon	[47]
Ada	Elf	[43]

Table 2.2. High Level Programming Languages in Silicon Compilation Systems

A behavioural description expresses the actions performed by the algorithm. Ideally it contains no implementation details that would suggest how the algorithm is to be supported in hardware. Instead it gives the reader an idea of how the data is processed, what the inputs and outputs are, and which operations are performed.

The algorithm is generally presented in some human readable format, usually a textual description. This might be expressed in French, Chinese or Newspeak - there are no limitations on the expression of the algorithm, only that it should be intelligible to the reader.

Therefore in the algorithmic description of digital systems it is important to use a language that is comprehensible in terms of digital components, but that does not restrict the design to a subset of the total components available. Of equal

[2]C with hardware constructs

importance is the ease with which the user should be able to write descriptions unimpeded by the language itself.

Programming languages are a popular choice for algorithmic descriptions. Being designed to be run on general purpose digital hardware they are generally intelligible in the digital sense. Their ability to deal with data of different sizes and dimensions make them a good choice, as do their constructs for data manipulation.

Examples of the use of programming languages include Flamel [6], HARP [45], ADA as a Hardware Description Language (HDL) [43] and the use of C within LAGER [48]. There is usually some restriction placed on the description language to ease the compilation task. This tends to downgrade the level of the synthesis, so that the language begins to resemble an HDL. Arguably this is no longer high level synthesis but logic synthesis. The work on ADA for silicon compilation [43] uses the language expressly for this purpose. Timing analysis data is also extracted from the description.

Within the DSP community applicative languages like Silage [35] are used for behavioural description. Silage is an applicative language that directly represents a data flow graph. It has no variable assignment as such, in that variables are not implicitly defined as memory locations, as in the case of procedural languages like C and Pascal. Instead the language treats input, output and intermediate values as simultaneous, ie as streams of values. This allows them to be referenced at any time. Many of the usual programming constructs are supported, among them size definition, casting and conditional and iterative control flow.

The main idea behind Silage is that it is completely independent of implementation details: the true parallelism of an algorithm is exposed by the DFG nature of the language. The authors claim [48] that this makes the language superior to procedural languages which contain an inherent ordering of operations. However, it has been shown that reduction of programming languages to control and data flow graphs is feasible and that this also exposes all inherent parallelism.

Other languages used for synthesis include VHDLand Verilog [49]. These provide both behavioural and structural constructs. They were developed specifically for hardware representation and provide many constructs for synthesis support. The V-Synth system [50] uses behavioural VHDL for input descriptions.

The VHDL is decomposed into tuples and various transformations applied. The output from the tool is structural VHDL, organised into process statements which are then synthesised by a back end program.

2.1.2 Internal Representations

The behavioural language description is compiled into one or more flow graph representations. This facilitates analysis of the algorithm being described and provides a conveniently implemented structure.

A section of high level code essentially contains two flow representations. Firstly, there is control flow which shows the passage of execution through the program: the ordering of operations. Secondly, there is data flow which shows the relationships between data values within the program: their provenance and consumption.

The use of control and data flow analysis has long been used in the analysis of program code [51]. It's application to high level synthesis is a powerful technique for the generation of data processing hardware and the necessary control logic that goes with it.

Trickey [6] uses a combined scheme of control and data flow graphs, called a DACON. As is common in software compilation [52], the program code is decomposed into tuples. These are single assignment, operator expressions that simplify later analysis. The tuples are then arranged into basic blocks, which are sequences of code that have a single entry and exit point. Control flow is expressed by arcs between basic blocks, which may themselves contain conditions for their pursuance. Data flow dependencies exist within the basic blocks and are used in conjunction with the implicit control flow of the tuple order to reveal the parallel potential of the operations within the block. Trickey represents his basic blocks as DAGs which are themselves nodes in the DACON.

The HYPER system [48] compiles it's Silage descriptions into a CDFG, a control/data flow graph. Data operations are represented as nodes within the CDFG. Edges between the nodes represent flow of control and data dependencies. Control edges can be added to the graph as a means of introducing precedence to the operations of the algorithm.

Macro substitution of iteration, function calls and conditional nodes in the CDFG with subgraphs for the respective blocks give the CDFG a hierarchical nature. This preserves the structure specified by the user and is a mechanism for supplying structural hints to the synthesis system [48].

This mechanism appears to be at odds with the aims of the Silage language in particular and algorithmic synthesis in general. However it is indicative of one of the problems facing high level synthesis; namely how to constrain the design space and arrive efficiently at a viable solution. Inevitably input from the user is required in some form or another, be it procedural groupings in Silage as hints for structural specification, or the use of pragmas to guide the allocation and instancing of hardware resources.

However necessary for synthesis, such annotations reduce the effectiveness of high level synthesis by forcing more of the detailed design decision making in the hands of the user, requiring a thorough knowledge of implementation issues. Whilst being well and good for the experienced and skilled designer, I contend that the target user group for high level synthesis should not necessarily require such knowledge in order to synthesise good design solutions.

2.1.3 Scheduling and Allocation

One of the most important tasks within high level synthesis is the scheduling of events within the design and the allocation of hardware resources to support these events. Paulin and Knight [16] suggested these classifications for the different methods used in scheduling and allocation: independent, interdependent and stepwise refinement.

Independent scheduling (that is independent of allocation), includes techniques such as As Soon As Possible (ASAP), As Late As Possible (ALAP) and list ordering. For these techniques the hardware allocation is fixed prior to scheduling. These methods thus lend themselves to iterative searches of a design space: hardware is allocated and it's suitability for the particular design can be gauged by the quality of the schedule produced.

Interdependent scheduling performs the allocation concurrently, resulting in a more guided search of the design space. These techniques tend to be based on

critical path analysis. Here hardware is generated and a schedule produced for the critical path of a design. An additional measure is required to schedule and allocate nodes outwith the critical path.

Stepwise refinement performs the grouping of events and allocation at the start, to give an initial scheme. This is then improved by the application of a function onto the scheme. Methods in this category include force directed scheduling, state splitting and clustering algorithms.

McFarland et al [28] use a similar classification that groups methods as either transformational or iterative/constructive. This classification cuts across the boundaries of those defined by Paulin and Knight. Transformational techniques start from an initial schedule and gradually alter it until resource or time constraints are met. Iterative/constructive techniques build the schedule step by step, attempting to select the next-best operation at each point in the process.. Examples of scheduling techniques are discussed below and placed in the context of these classifications.

ASAP and ALAP Scheduling

These two methods, As Soon As Possible (ASAP) and As Late As Possible (ALAP), are commonly used as sub-steps for some of the more complex scheduling algorithms. It is also commonly used in microcode scheduling, for minimising the number of machine states [53]. For example, ASAP and ALAP are used together in the determination of operator "freedoms" in the MAHA [54] system, and for the analogous "mobility" in SLICER [55]. Together ASAP and ALAP determine the period in which operations take place.

The two techniques are essentially the same, the only difference being the initial ordering of the schedule queue. For ASAP the order is given by the earliest point at which an operation can occur, concordant with the dependences involved. In the case of ALAP, the list is ordered according to the latest time at which an operation can occur. For both, hardware is pre-allocated before scheduling starts, so it is independent scheduling. The basic algorithm for both types of scheduling is shown in Figure 2.1

This scheme is relatively simplistic, it takes only local considerations into

```
Allocate hardware resources
Order operations based on the earliest time
they can be scheduled, place in schedule queue, SQ
Initialise control step counter while SQ != φ do
 pop operation o from SQ
 if hardware available to schedule o
  assign step[o] to step counter
 else
  push o back onto list
  increment step counter
 fi
od
```

Figure 2.1. ASAP Scheduling Algorithm

account which can lead to sub-optimal usage of the hardware resources. The method can be refined by prioritising the operations in the schedule queue, to ensure that groups of events connected computationally are scheduled together. This technique brings global constraints into the scheduling process, and is called List Scheduling.

List Scheduling

The usual method for List Scheduling is to apply some priority function to the unscheduled events at the start of each control step. The priority function usually selects the events on the critical path as having the highest priority. The critical path is defined as those events with the tightest constraints upon them. As mentioned above, MAHA [54] and SLICER [55] use the concepts of "freedom" and "mobility" to determine the critical path, this quantity being the difference between the event's ASAP time and ALAP time. Events with low movements are critical.

Other priority functions are used. "Urgency" is one which seeks to identify critical events. It is the distance from an event to the next constraint. The BUD system [56] uses a similar measure, the distance from an event to the end of the block in which it resides.

The HYPER compiler [57] utilises the concept of hardware urgency. At each control step two ratios are calculated for each type of hardware resource available. The first is a local constraint - that is the number of operations that can be scheduled in that step against the number of resources of that type available. The second gives the number of still unscheduled events of that type to come, against the resources available. The scheduling algorithm attempts to maximise the minimum ratio in each case. This technique will favour those operations on the most critical path, leading to optimal use of the available resources.

A computationally more expensive technique is force-directed scheduling [58], which tends to balance the distribution of different operations, resulting in better use of hardware. This leads to a minimal allocation. The concept of operation freedom, called the time-frame, is used again, except here it's value is taken to mean the probability of that operation occurring in a step within the time-frame.

A distribution graph for each type of operation is then created across the control steps. This is the summation of all probabilities in a time step for the particular operation. This graph indicates the concurrence of similar operations.

The next stage of the scheduling is to calculate the force on each operation in each control step. This is accomplished by performing the same calculation as before, except with the time-frame reduced to the step in question. The force exerted on the operation in that control step is the difference between these two summations. Operations are then scheduled into the steps which give the minimum force.

In [59], force-directed scheduling is applied to list scheduling. Operations are scheduled according to their control and data priority. When a conflict occurs the ready operations are prioritised according to the forces that act upon them. This avoids the computationally high costs of pure force-directed scheduling, whilst retaining the resource balancing benefits of the technique.

In [60] an inter-dependent scheduler is described. A hierarchical clustering algorithm is used for allocating hardware. This attempts to group hardware modules according to a number of criteria, including the quality of the schedule achieved. Expected freedom is used as a priority in the list scheduler. This quantity is a combination of the actual freedom of the operation and the probability that the current path is selected. The clustering algorithm that generates

the allocations performs grouping of separate clusters according to a minimum distance measure, derived from module parameters and schedule length.

Transformational Scheduling

Percolation-based scheduling [61] is a transformational method which is used for scheduling operations in the body of an iterative loop. Whereas the previously described techniques construct an optimal schedule in a repetitive manner, this technique attempts to start from an optimal schedule, and fits this to the allocation.

The loop body is iteratively unrolled, and operations within the code are "percolated" toward the start of the schedule, as far as dependences between them allow. This migration is limited only by the data dependences between the operations in the unrolled code, although the authors do not address the issue of array dependences. The method effectively pipelines the events in the loop, uncovering the overlaps between successive iterations.

After a small number of unrolls a pattern emerges between the repeating events in the schedule. This pattern represents the optimal schedule for the loop body. Once this has been established the mobility of each operation is found. This is used in binding the events to resources in the allocation, by providing a delay between conflicting operations in the maximal schedule. Delayed operations have their successors percolated upwards in the schedule, these may fit into gaps caused by the delay operation.

An extension of the technique deals with conditionals in the code description. The authors claim that the technique separates heuristics from the core of the algorithm, whereas others, for example in the case of loop folding [62], use heuristics for applying transforms during the scheduling. With percolation scheduling the final schedule is derived from the optimal case, reached without the use of heuristics, and therefore is provably optimal for the hardware resources available. I would contend that selecting operations to delay during the scheduling stage would involve heuristics and this might not always guarantee the optimal solution.

2.1.4 High Level Transforms

In this section the subject of transforms for high level synthesis is presented. Such transforms typically operate on the input representation. Depending on the type of transform, they are applied in either a global or a local sense. They are applied in order to improve the quality of the designs synthesised. However, as we shall see, it is difficult to establish whether a design's "quality" is improved through the use of such transforms. Some of the more common compiler transforms are reviewed below, these are then followed by array and loop-based transforms.

2.1.5 Common Compiler Transforms

Many high level transforms originate in the field of software compilation. They are well documented and known in the literature, and are used in a number of high level synthesis systems. These standard transforms are:-

- Constant Folding
  This transform involves the evaluation and substitution at compile-time of any constant expressions. These can include algebraic operations performed on constant values and identity operations introduced into the code description during the parsing of the program code.

- Common Subexpression Elimination
  Removes redundant operations by saving intermediate results that are used in later calculations. It is simply accomplished by subtree analysis between successive statements within a program description [52].

- Dead Code Elimination
  Removes code from a program description that is never reached during execution. It involves analysis of branch and procedure exit conditions [51].

- Loop Code Motion
  A form of common subexpression elimination within the lifetime of a loop [52]. Expressions (or sub-expressions) situated inside a loop body with operands that are not modified during its lifetime are removed outside the loop body.

The effects of these transforms are two-fold. In general they reduce the processing load on the hardware, the hope being that this will improve the execution time and also reduce the hardware complexity. A side-effect is to increase the register load in the hardware. Most require the storage of intermediate results and extra hardware might be required for this. Good register allocation, using techniques such as clique-partitioning, or graph-colouring will ensure that this extra hardware is minimised.

The CATHEDRAL II [36] system uses constant folding and subexpression elimination [23], as does the CMUDA project [63], for improving the input descriptions given by the user. Additional transforms applied by the latter system were

- Inline Procedure Expansion
  This transform substitutes the calls to a procedure by the procedure body, given in the definition. In software this transform gives a performance improvement when the code necessary to implement the procedure call is comparable with the actual procedure code, at the expense of increased code size.

- Loop Unrolling
  The code contained within a loop body, including operations performed at the end of each iteration, is duplicated and appended to the end of the original body. The code that controls the iteration steps is updated and conditional exit tests might need to be performed between the two pieces of code.

These transforms can increase the operator parallelism, for example by mixing the operations performed in a procedure body with the calling code, better use of allocated hardware might result. Loop unrolling also achieves this effect, as well as providing a technique for loop pipelining, see [7] and [61]. A similar technique is presented in [64].

Other transforms include the decomposition of operators: multiplication by constant powers of two can be replaced by shifts of magnitude equal to the power index. In the CATHEDRAL-2nd [18] environment multi-precision and floating point operations are decomposed into low level primitives. This type of transform

is more hardware specific than those mentioned previously, the gains are expected through use of cheaper or simpler hardware.

The transforms mentioned so far are essentially local in their effect, although they can have global consequences and can be used to ensure that expensive hardware is not allocated. For example, decomposing multiplications into series of shifts and additions will avoid the need to allocate a multiplier. This touches on the issue of choice of implementation and how particular operations are performed in hardware. This requires the application of heuristics based on a knowledge of the field of application.

Additionally some of the transforms might require reversal. For example decomposing a multiplication by a factor of two into a shift might, at allocation time, lead to a reduction in the hardware utilisation factor. A shifter is required to perform an operation that could be performed by an idle multiplier unit at that control step.

Trickey, in [65] and [6] uses various block level transforms in a global fashion. A transform tree is constructed which is traversed to find the best solutions to the synthesis task. The transforms used aim to increase the parallelism of operations at the basic block level. They are:-

- Line Merge
  Adjacent basic blocks are merged together, this is generally used after the application of the other transforms.

- Alt Merge
  Conditional branches are merged together.

- Unroll
  Unroll a loop once, doubling the body length.

- Full Unroll
  Unroll a loop body by the number of iterations of the loop - this value must be compile-time evaluable.

Trickey claims that the typical degree of parallelism in basic block code is 2 or 3 statements. Using the above transforms in the Flamel compiler increases this to between 5 and 10 statements.

The V-Synth system [50] performs many of the previously mentioned transforms such as constant subexpression elimination, code motion, dead code elimination. Expression factoring is also used, this orders individual operations in a complex expression. The claim is that this can reduce the required hardware resources. Inter-block parallelism is analysed and block coalescing, similar to that performed by Trickey, is attempted. The results show good hardware utilisation compared to designs that are synthesised without the transforms turned on. In [50] the authors claim a 25% average decrease in active area for synthesised designs for an average 62% increase in synthesis time.

2.2 Memory and Memory Architectures

In spite of the variety of synthesis systems available few address themselves directly to the issues of memory synthesis. The related topic of register allocation is dealt with in many systems, usually by use of clique partitioning as in the BRIDGE system [66], the ADPS system [67] and the Yorktown Silicon Compiler [68]. Other techniques include performing register minimisation during the scheduling phase of the synthesis process, as used in As Fast As Possible (AFAP) scheduling [69]. In this system lifetime analysis is used as a conflict resolution strategy for memory minimisation.

Register allocation and minimisation is an important activity within synthesising data paths. However before this can be carried out variables have to be scheduled onto data paths and decisions made about where these variables reside. This can be achieved in either global or local memory. Data in global memory must be addressed in the correct order and supplied to the processing unit or units at the right times. When large amounts of data are being dealt with at high speeds, as in the case of image and digital signal processing, the movement of data between these two areas of memory becomes critical to the performance of the synthesised hardware.

This type of synthesis domain has been characterised using the Hardware Sharing Factor (HSF). This quantity is defined as [70],

$$HSF = f_{clk}/f_{samp}$$

where f_{clk} is the frequency of the system clock and f_{samp} is the sampling frequency for the algorithm. High throughput applications have $10 \leq HSF \leq 1$, medium throughput applications have $100 \leq HSF \leq 10$ [70]. A further characterisation of the synthesis domain of high throughput algorithms is the number of operations per second that are specified. Typically this is $10M \leq ops \leq 100M$ [37].

It is in such domains that synthesis systems *must* address the access of data in memory. This is due to the large amount of data that is generally accessed and the high sample rate. Efficient memory management is the only way to make optimal designs in such domains. The following is a review of the two principal systems that address memory synthesis, and a summary look at some of the techniques they employ.

2.2.1 Memory Management in the CATHEDRAL Tools

In [18] attention is paid to the organisation of data in memory for the CATHEDRAL II system [71]. The reason for this attention is to improve the performance of the synthesized designs, especially in the case of those performing matrix type operations. It should be added that the inclusion of memory management functions in the design process is a result of the inability of the compiler to return efficient designs without it [18].

The same strategy is used in CATHEDRAL III [37], the latest IMEC compiler. These two compilers are similar in their aims. They both generate low multiplexed hardware consisting of separate datapaths and controllers interacting through memory blocks and register files. The datapaths generated by these compilers allow limited hardware sharing because of the relatively low HSFs (see above) of the target domain.

The compilers perform memory management on two "levels" of memory in the design environment. The first, upper level is the background memory for the hardware system, primarily blocks of RAM and Pointer Addressed Memories (PAMs). The second layer is referred to as foreground memory and is made up of the register files that are associated with the execution units.

The background memory management task determines the number and type of background memory units. It also selects the relative addresses for the data

in these memories. It ensures there is sufficient I/O support so that the overall performance of the hardware is not constrained by memory accessing. This is an important consideration for the application domain, as well as the realisation of efficient designs. Another aim of memory management is to minimise memory complexity and size. Given the physical space occupied by memory on and off chip, this is also an important task in the synthesis procedure.

Low level memory management in the CATHEDRAL tools is performed at a later stage of the synthesis process, after functional units have been allocated. It is primarily concerned with local variable organisation into registers and register files at the inputs and outputs of the functional units. The background memory management decides which variables are local and what type of memory they should occupy, thus the most important decisions regarding the memory are made in this first stage. Local memory management is more akin to standard register allocation, discussed above.

As mentioned in [37] and [18] many of the memory management techniques are still performed by hand, based on knowledge of the algorithms' behaviour. Whether this is achieved by updating the input script with the required structural information or by manually directing the memory management towards the desired structures is not indicated. Thus the solutions in the literature, for example [72], are assumed to be the result of hand application of the techniques described in [37] and [18]. The theory described in [18] does present some solutions to the problems of memory synthesis, and this is further expanded upon in [73] where a methodology is described for deducing the memory requirements of an ASIC solution is presented. An overview of these techniques is given below.

The signal flow graph taken from the Silage description (see Section 2.1.1) is examined for loops. These are listed in a priority order which is used to guide the memory synthesis. The list is ordered by the depth of nesting of loops (deepest first) and the maximum number of iterations. This makes sense as the innermost loops are critical, being executed the most.

Signals are then listed according to their relationship to the loop induction variables, that is whether they are defined or used in the loop statements. The signals on this list are then placed into either foreground or background memory. Some criteria have to be applied in this decision making:-

- If a variable is used within a certain number of loop iterations it can then be assigned to foreground memory. This threshold is related to the size of the register file at the input to the functional unit, which is usually 8. The threshold changes as signals are assigned to foreground memory in order to reflect the limited capacity of the register files.

- If this creation "distance" is greater than the register file threshold but smaller than some other, higher threshold then the variable can be assigned to a local memory: a FIFO or Pointer Addressed Memory. This leads to a local memory being created which can be used to access the data faster than from the main store. The higher threshold is related to the maximum permitted size of these local memories.

- When a signal is stored in background memory then addressing information is annotated to the input description. This will be used in later stages of the synthesis process.

In order to increase throughput, additional structures can be added to the memory hierarchy. These includes further shift registers and FIFOs that reduce I/O traffic between background memory and the processing core. Their inclusion involves manipulating the flow graph description of the input so as to find redundant accesses to the same data.

Employing these techniques leads to the construction of a memory hierarchy comprising mass memory, intermediate memory and register memory. This hierarchy increases the throughput of the hardware implementation, ensuring a valid solution for the input.

The background memory management of the CATHEDRAL tools takes advantage of the applicative nature of the Silage [35] language. This expresses inherently all parallelism in a description. The memory management in the tools then maps this parallelism into either the time or the space domain. Much of the management does not appear to be automatic but is guided by the user, who must be aware of the constructs in the description that can lead to a more efficient memory structure.

2.2.2 Memory Synthesis in the Phideo Compiler

As with CATHEDRAL, Phideo [74] targets limited shared hardware architectures for applications that have low HSFs. It relies on pipelining rather than multiplexing and uses local loop optimisations for minimising both memory and execution time for a given algorithm.

Central to Phideo is the concept of Periodic Operations (PO), which represent loops in the hierarchical flow graph used as an input to the tool. These are defined in terms of a sequence of operations, O, which contain a number of iterations, $N(O)$, taken from the specification of the algorithm. Each PO has a start time $S(O)$ and a period, $P(O)$. These latter two quantities are allocated by the scheduler. POs are treated as separate entities during synthesis and are an important part of memory synthesis in this tool.

Force directed scheduling [16] (and see Section 2.1.3) is used by Phideo for the scheduling and allocation of data paths. They are first specified by the functionality of the PO. Only the functional units are allocated at this stage. Estimations for the cost of memory are used in the distribution functions. This costing uses the maximum number of simultaneous accesses as an approximation to the number of memories required, with the maximum number of variables alive simultaneously as an approximation to the maximum memory size.

These assumptions appear reasonable, taken as they are from the precedences in the flow graph input. No transformation of this input that would improve these costs is mentioned in [74]. Transformation occurs during memory synthesis, utilising the delay streams taken from the schedule.

Once scheduling is complete data, address and control streams are extracted from the schedules. A stream is defined by a start time and a period. There are two types: a production stream, representing data output from a data path; and consumption streams, sequences of data loaded into a data path.

Delay streams are created from the scheduled POs. A delay stream is a production stream with a number of associated consumption streams. The memory allocation problem addressed by Phideo is the allocation of these delay streams to memories. A straight mapping is not always possible as memories have maximum sizes and access conflicts may occur between different delay streams, or within

the same one.

Phideo attempts to transform the delay streams until a mapping to memories is possible. It then minimises the total area by merging different memories together.

Phideo uses two types of transform. Delay tree extraction solves conflicts between consumption streams in a delay by duplicating existing streams. Individual consumption streams in a delay are re-composed as a delay with the original stream acting as the production stream for the new delay. This leads to delays with fewer consumption streams so there is less likelihood of access conflicts occurring. The technique aims to solve conflicts and minimise duplication of variables. All streams in the new delay tree existed prior to the application of the transform.

The second type of transform produces new streams in the delays. Decimation leads to streams with longer periods, which will effectively split data between different memories, thus reducing conflict.

Access time shifts produce two new stream delays. This third transform relies on the same data being reaccessed by the PO. Creation of a new stream will provide a small delay which will hold the reaccessed data.

Once conflict resolution has been accomplished then memories are mapped to the stream delays. Delays created by access time shifts will result in shift registers being allocated to implement the delayed-write, accelerated-read of data.

Once the memory has been allocated the tool attempts to minimise the total area by merging memories. The change in cost associated with each potential merge is calculated. Only those merges that lead to a decrease in cost are effected. This technique, having no hill climbing ability, may not encounter some optimal solutions.

Finally the addressing hardware is generated. Three addressing schemes are available: relative, counter and absolute. The allocated memories are dimensioned using each scheme and their final costs calculated. The scheme which gives the minimum total memory cost is selected. Sharing of address hardware is also attempted.

Phideo encompasses many of the requirements for synthesis in the high throughput applications domain. The importance of memory design is recognised

and addressed. The target architecture provides a memory hierarchy capable of breaking the access bottleneck, especially the use of delayed-write, accelerated-read components.

The optimisations performed by the tool will lead to good solutions. They are, however, very dependent upon good partitioning of the flow graph. No technique is put forward for achieving this, the flow graph is partitioned by hand. Thus periodic operations are grouped for maximal optimisation of the hardware before compilation begins. My contention is that high level synthesis should perform these functions automatically. In spite of this, the tool does allow for exploration of the solution space in an interactive manner.

2.3 Dependence Analysis and Program Transformation

Array access typically takes place in iterative loops in program code, with some logical conditions controlling the number of iterations of the loops. A typical construct in many languages uses a loop induction variable which is initialised at the start of the loop execution and tested against a given value before each iteration and altered at the end.

Much of the activity in array dependence analysis concentrates on such constructs, mainly in Fortran but the techniques are applicable to other languages. The attention to Fortran is due to it's widespread use in writing programs that benefit from fast hardware. However the techniques are gaining acceptance and with the spread of more powerful computers with greater functionality within a single microprocessor, software compilers are having to perform greater optimisation to fully utilise the hardware available.

2.3.1 Array Dependence Analysis

In order for one statement to be dependent on another they both have to access the same memory location at some point. With respect to scalar variables this means that each statement contains the same identifier, or identifiers whose variables refer to each other in some way. In the case of array variables the statements

are dependent if each one refers to the same location in the array. Thus two statements that refer to the same array are not necessarily dependent. Their indexes must be shown to be the same in order to confirm dependence.

For array statements in iterative loops the problem is generalised to detecting whether the two regions of memory accessed by the statements overlap. These regions are represented by the index expressions for each array, bounded by the values that the loop induction variables take.

Banerjee [75] and others [76], [77] developed practical techniques for detecting such conditions for use in optimising compilers. A short presentation of these techniques and the theory behind them is given below.

A Notation for Dependence

A notation for formalising what is called the dependence problem has evolved. The object is to discover whether two statements, S and T, are dependent. It is assumed that S and T access the same array and that the index functions for the two accesses are linear in the induction variables. Where the functions are non-linear, dependence cannot be practically determined and must be assumed.

Four different types of dependence can exist. For T being dependent upon S, written S δ T, these are,

- A *flow* dependence, where T reads the location written by S
- An *anti* dependence, where T writes the location read by S
- An *input* dependence, were T reads the location read by S
- An *output* dependence, were T writes the location written by S

In the notation these are represented as S δ T, S δ^* T, S δ^i T, S δ^o T, respectively. For dependences in loops the relationships can be *loop-carried*, where the access of S takes place in a *prior* loop iteration to that of T, or *loop-independent*, where the access of S takes place in the *same* one.

This presentation assumes a single dimensional array, though the results can be generalised to multi-dimensional arrays by applying the tests to each array dimension.

Call the set of loops that nest both S and T **L**, the set of loops that nest only S $\mathbf{L_S}$, and the set of loops that nest only T $\mathbf{L_T}$. The sets have members:-

$$\begin{aligned} \mathbf{L} &= L_k \text{ for } 1 \leq k \leq e \\ \mathbf{L_S} &= L_k \text{ for } e+1 \leq k \leq m \\ \mathbf{L_T} &= L_k \text{ for } m+1 \leq k \leq n \end{aligned}$$

The induction variables for these loops are i_k for $1 \leq k \leq n$. Each induction variable has bounds $p_k \leq i_k \leq q_k$ and it is assumed they increase incrementally by one.

Diophantine Index Functions

To illustrate the techniques used we will examine the case for the two statements S and T, where S comes lexically before T, that is S $\leq$ T and that S contains a write access and T a read, thus a test for flow dependence is performed. The index function for S is $f()$ and for T $g()$ which are linear diophantine functions of the nesting loop induction variables.

In order for a dependence to exist the relation $f() = g()$ must be true. Expanding the functions we get

$$a_0 + a_1 i_1 + a_2 i_2 + \ldots + a_e i_e = b_0 + b_1 j_1 + b_2 j_2 + \ldots + b_e j_e$$

where a_k, b_k refer to the function coefficients of the relevant induction variables. A simple case is when $e = 1$, i.e. there is a single loop nesting both S and T. In this case the equation becomes

$$a_0 + ai = b_0 + bj$$

with the condition that $p \leq i \leq j \leq q$. If an integer solution can be found to this equation then dependence is shown to exist.

From [78] the following results apply, setting $c = a_0 - b_0$, $b = -b$, $x = i$, $y = j$

we are to solve

$$ax + by = c \tag{2.1}$$

This is obtained by combining solutions to the related equations

$$ax + by = 0 \tag{2.2}$$

$$ax + by = d \tag{2.3}$$

where d is the greatest common divisor (gcd) of a and b. The general solution to Equation 2.2 is

$$(x, y) = (-b't, a't)$$

where $a' = a/d$ and $b' = b/d$. A particular solution (x_0, y_0) to Equation 2.3 can be found using an extension of Euclid's Algorithm for finding the gcd of two numbers [78], reproduced below.

1. $(x_1, y_1, c_1) \leftarrow (1, 0, |a|)$
   $(x_2, y_2, c_2) \leftarrow (0, 1, |b|)$
2. if $c2 = 0$ then
   $(x_0, y_0, d) \leftarrow (sig(a) \times x_1, sig(b) \times y_1, c_1)$
   and terminate
3. $q \leftarrow \lfloor c_1/c_2 \rfloor$
4. $(t_1, t_2, t_3) \leftarrow (x_1, y_1, c_1) - q(x_2, y_2, c_2)$
   $(x_1, y_1, c_1) \leftarrow (x_2, y_2, c_2)$
   $(x_2, y_2, c_2) \leftarrow (t_1, t_2, t_3)$
5. Goto 2

The *signum* function, $sig()$, used is defined by

$$sig(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

The general solution to Equation 2.1 is given by

$$x = c'x_0 - b't,\ y = c'y_0 + a't \tag{2.4}$$

where t is an arbitrary integer, and $c' = c/d$. This solution along with the bounds of the original diophantine equation can be used to obtain the values of the induction variable for which the dependence exists.

Dependence Testing

The simple example given above illustrates the methods used for dependence testing. From this result the simplest test for dependence is to take the gcd of the equation coefficients. If no gcd can be found then the index functions cannot fulfill the original condition $f() = g()$ and so no dependence exists between the two statements. This is the general pattern for dependence testing: it is assumed unless it can be proven otherwise. This is necessary as the test does not always find an exact solution.

This is especially true when dealing with general diophantine equations. In this case the gcd test is too general and will find a solution to the equation which is outside the bounds of the induction variables. To effect a more exact test it is necessary to apply the Banerjee inequality to the index functions. These inequalities were first used in [79] and were developed further by Wolfe [76] and Allen and Kennedy [80].

The Banerjee inequalities are approximate tests which use results from methods used to find the bounds of linear equations. As above, the examples given here are taken from [78]. For a diophantine equation, f in n variables where

$$f \equiv a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$$

in a region $\Re$ that is a rectangle, a solution exists if the banerjee inequality given by

$$b_{low}(f, \Re) \leq c \leq b_{up}(f, \Re)$$

holds. The rectangle is given by

$$x_k \in \mathbf{Z} : p_k \leq x_k \leq q_k : 1 \leq k \leq n$$

and the bounds are

$$\begin{aligned} b_{low}(f, \Re) &= \sum_{k=1}^{n} (a_k^+ p_k - a_k^- q_k) \\ b_{up}(f, \Re) &= \sum_{k=1}^{n} (a_k^+ q_k - a_k^- p_k) \end{aligned}$$

where the positive part z^+ and the negative part z^- of a number are as follows

$$\begin{aligned} z^+ &= max(z, 0) \\ z^- &= max(-z, 0) \end{aligned}$$

This inequality also holds for more general regions made up of a finite number of trapezoids. This allows for the determination of dependence in the case of inner induction variables initialised to those in outer nests. The bounds calculations are different to those above and are not covered here.

Before presenting the general algorithm for solving the dependence problem a further concept is needed. The *level* of a dependence refers to the depth of nest at which the relation is sought.

An algorithm for determining the existence of a dependence relationship between the two statements S and T is taken from [78]. The conditions mentioned previously (see page 2.3.1) are assumed. Furthermore a direction vector $\mathbf{s} = (s_1, s_2, \ldots, s_e)$ is required where $\mathbf{s}$ is determined by the conditions placed on the induction variables of the nesting loops.

$$s_k = \begin{cases} 1 & \text{if } i_k - j_k < 0 \\ 0 & \text{if } i_k - j_k = 0 \\ -1 & \text{if } i_k - j_k > 0 \end{cases} \qquad 1 \leq k \leq e$$

Appropriate vectors for **s** are selected for S and T, and the algorithm detailed in the next section is used to determine if such dependences exist.

The Banerjee Inequality

Given a direction vector **s**, which shows the relative positions of the two statements, S and T, in the loop lifetime, with the index expressions for S and T being

$$a_0 + a_k x_k$$

and

$$b_0 + b_k x_k$$

for

$$1 \leq k \leq e$$

then dependence can be determined by:-

1. Partition the set of loops $\{1, 2, \ldots, e\}$ into four pairwise disjoint subsets

$$\begin{aligned} \mathbf{P_0} &= \{1 \leq k \leq e : s_k = 0\} \\ \mathbf{P_1} &= \{1 \leq k \leq e : s_k = 1\} \\ \mathbf{P_{-1}} &= \{1 \leq k \leq e : s_k = -1\} \\ \mathbf{P_*} &= \{1 \leq k \leq e : s_k \text{not defined}\} \end{aligned}$$

2. The diophantine equation to be solved is

$$\begin{aligned} lll \sum_{k \in \mathbf{P_0}} (a_k - b_k) i_k + \sum_{k \in \mathbf{P_1}} (a_k i_k - b_k j_k) + \\ \sum_{k \in \mathbf{P_{-1}}} (a_k i_k - b_k j_k) + \sum_{k \in \mathbf{P_*}} (a_k i_k - b_k j_k) + \\ \sum_{k=e+1}^{m} a_k i_k - \sum_{k=m+1}^{n} b_k j_k \\ = b_0 - a_0 \end{aligned}$$

3. Find the gcd, d, of the set of numbers

$$\begin{aligned}
(a_k - b_k) \quad & \text{for } k \in \mathbf{P_0}, \\
a_k \quad & \text{for } 1 \leq k \leq e \wedge k \notin \mathbf{P_0}, \\
a_k \quad & \text{for } e+1 \leq k \leq m, \\
b_k \quad & \text{for } 1 \leq k \leq e \wedge k \notin \mathbf{P_0}, \\
b_k \quad & \text{for } m+1 \leq k \leq n,
\end{aligned}$$

   if $d = 0$ or d does not divide $b_0 - a_0$ then the test fails

4. Check the inequalities,

$$\begin{aligned}
& \textstyle\sum_{k\in\mathbf{P_0}}(-(a_k - b_k)^-(q_k - p_k) + (a_k - b_k)p_k) \\
+ \; & \textstyle\sum_{k\in\mathbf{P_1}}(-(a_k^- + b_k)^+(q_k - p_k - 1) + (a_k - b_k)p_k - b_k) \\
+ \; & \textstyle\sum_{k\in\mathbf{P_{-1}}}(-(b_k^+ - a_k)^+(q_k - p_k - 1) + (a_k - b_k)p_k + a_k) \\
+ \; & \textstyle\sum_{k\in\mathbf{P_*}}(-(a_k^- + b_k^+)(q_k - p_k) + (a_k - b_k)p_k) \\
+ \; & \textstyle\sum_{k=e+1}^{m}(-a_k^-(q_k - p_k) + a_k p_k) \\
+ \; & \textstyle\sum_{k=m+1}^{n}(-b_k^+(q_k - p_k) - b_k p_k) \\
\leq \; & (b_0 - a_0) \qquad\qquad \leq \\
& \textstyle\sum_{k\in\mathbf{P_0}}((a_k - b_k)^+(q_k - p_k) + (a_k - b_k)p_k) \\
+ \; & \textstyle\sum_{k\in\mathbf{P_1}}((a_k^+ - b_k)^+(q_k - p_k - 1) + (a_k - b_k)p_k - b_k) \\
+ \; & \textstyle\sum_{k\in\mathbf{P_{-1}}}((b_k^- + a_k)^+(q_k - p_k - 1) + (a_k - b_k)p_k + a_k) \\
+ \; & \textstyle\sum_{k\in\mathbf{P_*}}((a_k^+ + b_k^-)(q_k - p_k) + (a_k - b_k)p_k) \\
+ \; & \textstyle\sum_{k=e+1}^{m}(a_k^+(q_k - p_k) + a_k p_k) \\
+ \; & \textstyle\sum_{k=m+1}^{n}(b_k^-(q_k - p_k) - b_k p_k)
\end{aligned}$$

   If these inequalities do not hold then the test fails, otherwise the test succeeds.

If the test fails then the dependence being sought does not exist. If the test succeeds then there may be a dependence of T upon S with direction vector **s**

and we have to take it as given.

Dependence Distance

The test described in the previous section is not an exact test, it guarantees that no dependence exists between S and T if the conditions are not met. However, when dependence is indicated, that is both conditions are met, then an integer solution is not necessarily guaranteed. In [78] the author states that if the distance exists then all coefficients in the equation are from the set {0,1,-1} then dependence is guaranteed, that is an integer solution to the equation exists.

Usually a number of solutions will exist, each represented within the iteration vector pair $((\mathbf{i}_k),(\mathbf{j}_k))$. If the dependence distance for each pair, $(\mathbf{j}_1\text{-}\mathbf{i}_1,\ldots,\mathbf{j}_k\text{-}\mathbf{i}_k)$ is the same then the distance vector is said to be *constant*.

Testing of Symbolic Bounds

Allen [77] extends the Banerjee inequality by considering symbolic bounds for the induction variables. These are most likely to be upper bounds, where a loop induction variable is specified as

```
for (i = 0;i < N;i++)
```

and N is a symbolic quantity. Normalisation of loop bounds further renders this case the most common.

In the gcd test the bounds do not feature at all, thus if it fails there can be no dependence, and the existence of symbolic bounds is irrelevant. However, given the weakness of the gcd test this occurrence is insignificant in most cases.

The Banerjee test makes repeated use of the bounds as shown from page 39. Allen suggests that examination of the terms in the inequality shows that the effect of a symbolic upper bound will only decrease the value of the left-hand side and increase that of the right-hand side. This can be seen by examining the first term of the left-hand side:-

$$\sum_{k\in\mathbf{P_0}} (-(a_k - b_k)^-(q_k - p_k) + (a_k - b_k)p_k) \tag{2.5}$$

Here the upper and lower bounds are given by q_k and p_k respectively. Assuming that $q_k > p_k$ (ie after normalisation), and remarking that the term $-(a_k - b_k)^-$ will always be either negative or zero, then the effect of a symbolic upper bound will always make the left-hand side more negative. A similar examination of the right-hand side shows that the effect of a symbolic upper bound makes the equations more positive.

These observations enable Banerjee's inequality to be used for determining dependence in the presence of symbolic upper bounds. By first computing the inequality for all terms not containing a symbolic quantity, a partial result can be obtained.

If this result implies dependence then the statements must be assumed to be so. If the partial result implies independence then the symbolic terms are checked for zero coefficients; eg for $(a_k - b_k) \geq 0$, in the above case.

If any symbolic coefficient is found to be greater than zero then dependence must be assumed, as the symbolic term can only increase the magnitude of the equation. If all such terms evaluate to zero then independence is assured.

In the situation above where symbolic coefficients are found to be positive then an inequality involving the symbolic term can be obtained. This would give a bound on the value of the symbolic quantity and could be used to prompt a user for further information on the likely value of the upper bound.

Other Symbolic Terms

Allen also discusses additive symbolic terms in the index expression of the array. He shows that dependence can be determined for those symbolic terms that remain constant during the lifetime of the loop. Constant propagation (see Section 3.4.4) improves the handling of such cases.

For purely symbolic cases he concludes that the difficulties lie not with the tests themselves, but are inherent in the process of dependence determination. Symbolic data affects the dependence functions, indicating that unless their values are known their behaviour cannot be deduced.

These conclusions delineate the use of dependence analysis. In the presence of control and "controlled" data (such as loop induction variables) dependence

can usually be determined. When "uncontrolled" (such as statically, or compile time, un-evaluable) data is involved then dependence analysis has little to offer.

2.4 Further Transforms

The previously reviewed (Section 2.1.4) high level transformations used in a number of synthesis systems have been well known to compiler writers for quite some time. Similarly the high level synthesis community has been using them for a number of years [81]. Many of these transforms are useful in tidying coded descriptions and normalising human readable descriptions to descriptions more appropriate for specifying hardware. Some transforms, like loop unrolling, go further than this and change the actual behaviour of the description, whilst keeping it functionally true to the original.

It is this last type of transform whose application truly results in high level synthesis. However it is the application of such transforms that require more investigation [28], and the combination of such methods that will provide the means to perform *higher* level synthesis. With this in mind new types of transform are required. This section covers some that have been developed by the parallelising compiler community.

Much of this work has taken place in the last fifteen years or so [75, 82, 83]. The main impetus behind the research has been to exploit the parallelism found in ordinary programs on highly parallel machines. The reasoning has been that, rather than software writers learning new (parallel) languages to run on supercomputers or massively parallel machines, they should be able to write code using the traditional programming languages [84]. It is then left to the compiler to produce the optimised object code that will run on the machine.

Since much of the work carried out by scientific programs is done in loops, it is these constructs that the optimising compiler writers concentrate on. Parallelism in loop execution can be exploited in a multiprocessor environment by farming iterations out to different processors. To this end a series of high level transformations have been developed to isolate such parallelism in the source code. Some of these are reviewed here.

As well as loop constructs the transforms analyse the array accesses within

the loop bodies as a means of identifying parallelism. The techniques used to do this have a theoretical framework, dependence analysis. A short introduction to the notations used is presented first.

2.4.1 Array and Loop Based Transforms

Much of the optimisation in software compilation is based on the analysis of scalar variables in the program. This is also true in the hardware synthesis community. The transforms discussed in the previous section introduced a number of operations that are accepted by both communities as useful operations for optimisation that can be performed at a high level. In the main, these all involve scalar variables. The recognition of array accesses is not generally incorporated in these transforms.

The main reason for this is that dependence analysis techniques that have latterly been used in optimising compilers have not been powerful enough to deal with array accesses and their interdependence. However, with the advent of massively parallel machines there has been much development in optimising compilers. These allow programmers to use traditional languages for the new architectures, eliminating the need to learn new parallel languages, or rewrite existing programs. Vector machines have also played a part in this development.

As a consequence much work has been directed at transforming source code in languages such as Fortran and C into code that can take advantage of the parallelism available. The code constructs that benefit most from such optimisations are loops, hence the application of these new optimisation techniques has been directed at loop structures and array accesses within them.

Section 2.3 reviews the work on dependence analysis for arrays within loops. The results of such analyses are used for many of the transforms presented here.

Simple Array Analysis

Array accesses are expensive, both in software and hardware terms. In software reading or writing data from an array will typically involve upwards of three separate machine instructions, depending upon the mode of addressing used: load the base address for the array, load the index value into the array, load the

data pointed to by the base and index.

In hardware the overhead is similar in terms of microinstructions. In both cases the time overhead is significant, especially as arrays are on the whole accessed from within loops so this sequence is repeated many times. This is compounded by the time involved in accessing memory from processing core, especially when the two parts reside on separate ICs.

Increased computing power and the move from CISC to RISC technology with the associated reliance on optimising compilers means that such optimisations are becoming common in compilers for general purpose machines.

A naive treatment of array accesses would be to recognise multiple references to the same array location within, say, a basic block. This would result in the code example shown in Figure 2.2

```
a = b[i] + d[i + 1];
c = b[i] * k;
```

```
t1 = b[i];
a = t1 + d[i + 1];
c = t1 * k;
```

Figure 2.2. Simple Code Example

This is a form of common subexpression elimination performed for array accesses. In [85] a method for detecting such situations is given. The technique is called *scalar replacement* and can be used as a front end technique for common compilers which will deal intelligently with allocating scalar variables to registers, but not array references. In this sense the optimisation is a source to source transform, the effect of which is to improve the performance of the final object code.

The basic method involves utilising loop-independent dependences and dependences carried by the inner loop (see Section 2.3.1). Dependences have *sources* which connect to *sinks* via an edge in the dependence graph. Sources and sinks are loads or stores from or to the same array. The edges between them indicate that they access the same location in the array. Replacing sinks with temporary variables assigned to the source value removes an array access from the code.

If a dependence is carried by the inner loop then a number of temporaries

will be required. This number will be determined by what is called in [85] the threshold of the dependence edge e, $\tau(e)$, which is the number of iterations between the source and the sink for a particular location. The number of temporaries required for each dependence edge is $\tau(e) + 1$.

References are then replaced by temporaries. If the source of an edge is a load then the temporary T_i^0 is assigned to the array expression. This temporary then replaces the reference to the array in the source statement. The references in the sink statements are replaced by the variables T_i^τ

Loop Transforms

Rearrangement of loop structures can lead to significant optimisations in software. As mentioned previously much of the study into parallelising serial program code has concentrated on the behaviour of loops. A review of the techniques that exist in the literature is given here.

Loop Interchange Loop interchange is the swapping of tightly nested loop statements. The benefits of this transform vary according to circumstances. In virtual memory systems accessing arrays in a column-row order might require a that new page be loaded into memory for each iteration of the inner loop. Accessing the array in row-column order would prevent this requirement. Analysis of the array dependences is required in order to confirm that such an interchange is necessary.

2.4.2 Cycle Shrinking

In order to parallelise loop constructs it is necessary to analyse the dependences that exist between the statements contained within the loop body.

A method for parallelising given in [84] is cycle shrinking. This method is used for parallelising loops whose bodies contain flow dependences with distances greater than one. The basic method is to find the smallest flow distance within the loop body and create a new parallel nested loop which executes this number of times within the outer, serial loop.

The method covers both simple and complex loops with constant dependence

```
for (i = 0;i < N;i += 1)
{
  a[i+k] = b[i] - 1;
  b[i+k] = a[i] + c[i];
}
for (i = 0;i < N;i += k)
{
  par_for (j = i;j < i + k;j += 1)
  {
    a[j+k] = b[j] - 1;
    b[j+k] = a[j] + c[j];
  }
}
```

Figure 2.3. Cycle Shrinking for a Simple Loop

distances. A simple loop has a dimension of one; a complex loop has more than one dimension. A loop is taken to have constant dependence distances between iterations, such that the array subscript expressions are of the form $i \pm a$ where a is some constant value.

The inner loop in the transformed part of Figure 2.3 is executed in parallel as there are no dependence conflicts within the execution.

2.5 Summary

This chapter has examined the areas of study relevant to the work presented in this thesis. An overview of the field of high level synthesis has been given, and some existing systems introduced. Issues within the field have been discussed, with the various approaches taken by different groups examined.

Many of these systems embrace a script-based paradigm which takes an input description expressed in a high level language and converts it into an internal format. The choice of language is fairly arbitrary, to an extent it is dependent upon the target domain; as, for example, in the case of Silage [35]. But, on the whole, the differences between the expressive power of the various languages is

fairly minimal. All support the same constructs and operations, input descriptions tend to be very close to the RT level for the design, the type of description a vanilla programming language with no pointers, or arrays, could support.

The internal format provides a suitable representation on which to extract information for synthesis. This is usually some form of flow graph, which may be pre- and/or post-processed for optimisation. Relationships between operations are easily expressed in such a form.

Amongst the most important tasks in high level synthesis are the scheduling of operations in time and the allocation of physical resources for the design. A number of techniques are popular and prove effective, this has been an area of great activity for some years, reflected in the range of methods available.

Increasingly, high level transformations are applied to the internal representation, in order to improve the performance of the synthesised designs. This provides a means of supplying specialist knowledge to the design process, removing responsibility for detail from the designer, and allows the solution space to be explored.

In keeping with the central theme of this thesis, the subject of memory synthesis is reviewed. This provides a comparison of the few systems that address this issue. Their reasons for doing so are domain specific. In the case of the CATHEDRAL II tool, a second version was developed in order to provide a memory synthesis capability that allowed the algorithms being looked at to be synthesised within performance constraints.

Memory accessing has been looked at in the field of compiler development, particularly for parallel machines, although increasingly for orthodox, serial machines. The subject of dependence analysis was introduced, with a notation that is used throughout the thesis. This analysis has an important role to play in the synthesis of memory architectures, and could be a key technique in the development of system level synthesis. Languages employed by high level synthesis systems could greatly benefit from their extension to include multi-dimensional arrays and their access from within loops.

Finally a number of transformations, based on dependence analysis, are presented. These come largely from fields outwith that of high level synthesis, but have important contributions to make, as will be shown in later chapters. The

transforms perform at two levels, inter-statement and intra-loop. These can have a significant effect on the performance of memory communication, when applied to hardware synthesis.

Chapter 3

Introduction to CSiC

This chapter presents some of the work completed during this research. It introduces a software program that has been written in order to perform memory synthesis from algorithmic descriptions. The program is called CSiC, for C to Silicon Compiler. Its main task is to support the analysis of such descriptions written in the C programming language.

The theory and discussion of the high level tasks implemented within CSiC are presented in Chapters 4 and 5. In this chapter some of the methods used in the implementation of CSiC are presented. Their uses are then explained in the later chapters. Some example input descriptions are used in an illustrative manner here and in these following chapters. Larger and more complete examples are given in Chapter 6.

The methods are presented in some detail with a discussion of what alternatives are also available. An examination of the requirements that led to the compiler's development is given, followed by a presentation of how these requirements are met in the tool. The general outline given in the introductory chapter to this thesis is expanded and developed further here.

3.1 Introducing CSiC

In order to analyse algorithmic descriptions for hardware synthesis a tool set is required. Such a collection of tools must be capable of performing a number of

different tasks. These tasks are: data input, data structuring, data manipulation and data output.

As the intention of this project is to make use of algorithmic software descriptions a compiler is the obvious choice as the basis of the software. The parsing of software code into data structures is a relatively mature technology [52]. The representation of software program code as data structures and the manipulation of these structures has also been thoroughly studied [51].

The analysis of the memory requirements and the synthesis of memory structures is the main goal of the project, the tools aim to support these aims. Such analysis requires attention to array accesses within the input description and also to the iteration, or loop constructs. The tool should therefore provide support for the representation and manipulation of these structures within the input description and also the relationships between them.

These requirements are met by compiler technology. A parse tree [52] structure can be used to represent the input description. This structure can be manipulated, transformed, changed and also used to regenerate altered code for verification purposes.

A fully functioning compiler is not required: there is no need to generate a full microcode description of the input code; the full arithmetic functionality of the input is not required, it is primarily the input/output behaviour of the code that interests us here, rather than the processing requirements.

The information to be extracted and output from the input description is a structural description with additional timing and dependence data.

Thus the support requirements of CSiC can be summarised as: transformation of software input into a regular data structure upon which transformations can be applied and from which can be extracted array, dependence and loop data, which together are used to generate a structural hardware description for the input code.

3.1.1 An Outline of CSiC

CSiC takes a hierarchically flattened input description written in C code. The hierarchy flattening is achieved by inline expansion of function calls within the

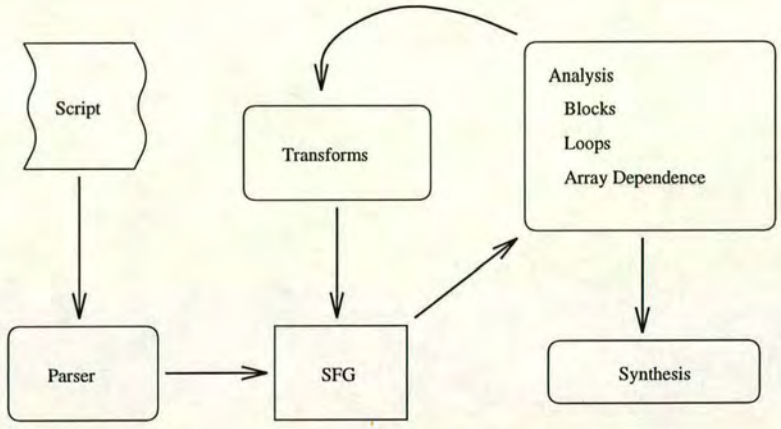

Figure 3.1. Overview of the CSiC program

input description. This removes calls to procedures within the program by macro expanding the procedure description at the point of calling. The program text is then passed to the CSiC tool.

The front end parser performs two basic functions: syntactic and semantic error checking for the input description, and generation of the internal data structure that represents the input description. Once the parsing process is complete, various cleaning tasks are performed on the internal data structure. These tasks aid the later stages by normalising the description.

Various analyses then take place which build up the information held about the input description. These stages include control and data flow analysis which gather relevant data to indicate possible orders of execution for statements in the program. The statement code is broken up into basic blocks which are used for subsequent analysis. A basic block is a piece of straight line code with a single entry and exit point. This is a standard technique used in software compilers [52].

Once the basic analysis is complete the program begins synthesis. This is basically an iterative process comprising extraction of architectural descriptions, evaluation of these descriptions and transformation of the internal representation, in order to extract better architectural descriptions. The bulk of this stage is covered in Chapters 4 and 5.

3.1.2 Coding of CSiC

CSiC is written in C++, an object orientated language [86] that supports abstract data types, multiple inheritance and dynamic binding. There are a number of different data structures used within the program, and many of these tend to be of the same general type. A prime example is the graph type, of which there are three main varieties. The use of C++ in this project has greatly eased the programming task by enabling a generic graph type to be defined, supporting functionality common to all the graphs. This generic class was then used as a base class for the main graph types, the resultant graph containing additional functionality particular to the task for which it is used.

The code is compiled using the GNU project compiler gcc [87] which provides good portability and reliability. The system currently runs on both a Sun workstation and a PC, and it comprises approximately 18000 lines of code and comments.

The excellent LEDA [88] library of standard types has been used. This greatly reduced the amount of work involved in the development of the software. LEDA provides many different parametisable types, including lists, sets, hash arrays, graphs and windows. Standard graph algorithms are also provided in the library which have proved useful.

3.1.3 Alternatives Strategies

The writing of CSiC has involved the development of a front end parser for the C language and the implementation of compiler transforms, optimisations and analysis stages. Much of this effort could arguably have been avoided by utilising software available in the public domain, or from other sources.

For a number of reasons, which are expanded upon below, this course of action was not taken. CSiC was written from scratch, primarily because code from disparate sources would have to have been combined in order to provide the functionality required. This would have been a not inconsiderable task in itself. The following sections examine some of the other systems considered for the development of CSiC .

Tiny and Omega

Wolf's tiny compiler [89] was placed into the public domain in 1990. It reads programs written in a pseudo-Pascal language and performs data dependence analysis. A number of restructuring transformations can be applied interactively to the input description.

An extension of tiny is the Omega test [90]. This builds on top of tiny, implementing further tests and transforms, notably the Omega test itself [91]. The latter is an integer programming technique for computing exact dependence information.

The Omega test software provides a good basis for dependence analysis for subscripted variables, using a vanilla input for testing transforms. It is aimed at the developers of compilers for vectorizing machines. For the purposes of CSiC, there are a number of shortcomings in the tool. These are primarily concerned with the input language and the constraints placed on it. These, however, would have further ramifications if the tools were used for CSiC .

CSiC is intended to use "real" software algorithms that are demonstrably functional on their input data. The examples available to the project are written in C and would have to have been translated into the pseudo-Pascal language used in tiny. Whilst this would not have been a huge task, there would have been further obstacles.

The pseudo-language provides no support for typing or sizing of data. Whilst not critical for CSiC this information is useful for costing of solutions that are synthesised.

Both tools have no concept of a variable, excepting arrays and loop induction variables. All other variable names are assumed to be identifiers for constant values. As this assumption is used in calculating the dependences this would inevitably have led to errors in the analysis of the examples. Altering the tools to remove this assumption would have involved a major rewriting.

Much of the preprocessing of input descriptions required by CSiC, covered in Section 3.4 below, would have to have been implemented as none of this support is provided in the tiny/Omega combination. This would have represented a further major rewrite of the software.

The Amsterdam Compiler Kit

This [92] package provides tools for building a front end parser for a number of different languages, such as C, Pascal, Ada. A major aim of the tool is to provide a standard platform for the development of portable compilers. The front end parsers generate an intermediate language which "runs" on a virtual stack machine. The objective here is to reduce the work required in developing new compilers for different computers to that involved in translating from the intermediate code to the native machine language.

Optimisations are performed on the intermediate language. This would be a major drawback for CSiC, because of the high level nature of the high level transforms used in CSiC, it is easier to verify their effects at the source as level, as this is where they operate.

The support provided for internal representations and optimisations would have been useful in the development of the CSiC tool. This would have saved some labour during the early stages of CSiC development, but as much of the design effort has been concentrated on the analysis and transformation stages, much of the software written would have had to have been implemented anyway.

Other Compilers

Gcc [87], and other compilers like it (lcc [93]), which are supplied free with full source code, have gained widespread support and use over the last few years. They are designed for software compilation over a wide range of platforms. This is accomplished by the use of intermediate representations, particularly in the case of the Register Transfer Language (RTL) [87] used in gcc. Again, these would be useful tools for constructing a front end to a compiler, but much of the back end required for silicon compilation would have to be written from scratch.

Some of the requirements of the project would have been met in part by one or more of the systems above, but much of the work accomplished would have to have been done anyway. In addition, the learning curve that was climbed by starting from a bare parser and designing an intermediate structure would still have been an issue, albeit with the examples of others.

3.2 Initial Stages of CSiC

As laid out in Section 3.1.1, CSiC has four main phases of compilation: the parsing of an input description; the analysis and collation of information contained in the input description; synthesis and evaluation of memory organisations for the input description; transformation of the internal representation using evaluation reports and information held within it.

The first of these phases is performed once only. The algorithm expressed in the input is converted into an internal representation suitable for processing by the succeeding phases.

It is intended that these perform in an iterative manner. The first extracts information from the internal representation to be used by the succeeding phases. This information comprises quantitative data required in the synthesis phase of the cycle, and also qualitative data which outlines various options for changes that can be made to the input description. This is used during the transformation phase of the compilation, which attempts to improve the quality of the solutions obtained by the tool. After this final stage the entire process, excluding parsing, is repeated. This iteration continues until the end conditions are met.

The rest of this chapter describes the first of two these phases, the initial parsing and the analysis performed on the input description. Synthesis and transformation are dealt with in the following chapters.

3.3 The Input Description

The starting point for synthesis is the input description, containing an algorithm coded in C that the user wishes to realise in hardware. The intention is that the algorithm be developed first in software so that information necessary to the hardware design task can then be extracted from the description.

This design methodology is widely used for embedded systems and is particularly suited to the image processing field where software prototyping is a flexible development tool for algorithm design. A number of designs known to the author have been conducted in such a manner [20, 94, 95].

The class of algorithms encountered in the field of image processing tend to be

iterative in nature and operate on fixed sized arrays of data. A typical application would involve a collection of such algorithms applied in some order and using a number of primary data stores, or arrays as for example in [27].

Some coalescence of the algorithms contained in such a description will be possible, as each sub-algorithm will have been considered separately by the software designer. Of course the programmer may have already coalesced the algorithms to some degree for the sake of efficient software. This however runs against the developer's main preoccupation, to ensure correctness of the algorithm as a whole, rather than to expend energy (and some ability to verify), on ensuring an efficient software implementation.

Once the algorithm is stable the program model can be optimised. It is at this point that initial hardware synthesis can begin using CSiC as a tool. True high level synthesis begins from an algorithmic description with minimal implementation information. It is to this end that CSiC has been developed. The aim of the tool is to increase the designer's productivity by facilitating the development of algorithms rather than concentrating on implementation level details. As much of the latter task as is possible should be devolved to the synthesis tools in order to achieve this goal.

So that these aims are realisable some constraints are placed upon the language components used in the input description. These constraints are largely commensurate with the expression of the class of application being addressed, namely image processing. Some restriction of the algorithms expressible will inevitably occur, however this is not necessarily detrimental. This issue is discussed in the next section.

3.3.1 Constraints on the Input

The restrictions that CSiC places on the input language fall into two categories. The first is concerned with the use of dynamic memory in program code; the second affects the analysis of dependence information. I deal with the two issues separately here.

A major difference between software and hardware realisations is the question of storage allocation. Virtual memory used by modern operating systems

offers potentially unlimited data storage for the programmer. Memory in general computing systems is becoming less of a constraint. The widespread use of high level programming languages which support dynamic allocation of memory leads to programmers relying more and more on the ability to grab storage when it is required.

The hardware designer on the other hand, does not usually have the support of an operating system in the hardware itself. Resources, and particularly memory, are finite and bounded. Performance is much more of an issue, a hardware solution is often preferred on the basis of an increase in performance over software. Additionally the designer is often working within cost constraints that demand a minimal hardware configuration, especially of memory.

In order to synthesise hardware from software these issues need to be addressed. Use of dynamic memory allocation must necessarily be disallowed in the input description of CSiC . A system level synthesis tool would be required to deal with such constructs, managing the organisation of data in the hardware domain as a whole.

For the same reason recursion is disallowed in the input. There is no reason why in theory recursion should be excluded, as long as the maximum depth attainable by a recursive procedure is statically evaluable, or some guarantee is given by the designer that such a limit exists. This however is beyond the scope of CSiC at the present time.

The previous two points constrain the input to CSiC to a set of algorithms, namely non-recursive procedures that operate on fixed size quantities of data. These constraints certainly exclude large classes of algorithm. However many of these fall outside the domain of image processing, much of which is concerned with the iterative processing of data taken from storage arrays or read from an input stream [96].

The second category of constraints pertains to the evaluation of data and control dependence within the program code. Specifically it affects the use of pointers and also the bounds of loops structures in the input.

The use of pointers in languages such as C is a powerful programming technique which can also have an effect on performance compared with the use of

array variables. However it complicates data flow analysis and leads to "conservative errors" [52]. Such errors will never affect the semantics of the program being compiled but may lead to suboptimal optimisations. In practice C places very few restrictions on pointer use. It would be possible to restrict the language in order to increase the optimality of conservative errors, but this could arguably no longer be classified as C.

Similar arguments exist over the use of "goto" constructs in code. These statements reduce the effectiveness of an optimising compiler and can make code less readable. In order to reap the benefits of optimising compilers it is generally recognised that use of gotos is not a good idea unless something specific is required. A similar argument can be put forward over the use of pointers. Given the inter-changeability of pointer and array expressions in C, why use pointers when array notation can be used instead? In answer to that, there is no advantage except in terms of software performance, unless the demands of the code require that pointers be used. In this case such an algorithm is probably too complex to be compiled into hardware automatically in any case.

With respect to image processing applications, the use of arrays is very applicable as the algorithms tend to operate on fixed sized data structures and access them in a regular manner. Thus pointer usage can mostly be replaced by array statements. This leads to more easily optimisable code which makes better hardware.

In order to be able to test for dependence between array accesses it is necessary to know the bounds of loops at compile time. This is a reasonable restriction on those loops that access image type data, which will mostly be bounded by the dimensions of the image data. For other loops this restriction may not be as reasonable. As mentioned in Section 3.5.1, it is possible to perform dependence tests on arrays nested by loops whose induction variables are initialised by outer induction variables.

Furthermore, in Section 2.3.1 Allen's methods for dependence testing with symbolic upper bounds are reviewed. These two results show that this last restriction is not as necessary. At present CSiC does not perform any of these types of test, therefore it requires that loops be statically bounded in order for dependence information to be retrieved.

These final restrictions could be ameliorated by implementing Banerjee's test for trapezoid regions [78] and extending the existing tests in CSiC for testing symbolic upper loop bounds [77]. These tasks would not represent a large task, but have no relevance to the examples used by CSiC at present.

3.3.2 Inlining

The input contains the algorithm coded in C. All user defined calls are inlined in a preprocessing stage using a commercially available inliner [97]. This flattens the procedural hierarchy of the input description, bringing it closer to the eventual hardware representation.

Procedural descriptions are primarily a software structure that aids code readability and allow the programmer to design in a modular fashion. This modularity is not necessarily helpful in the design of custom hardware. This is because resources for custom hardware are allocated with regard to the associated cost, balanced against the resources demanded by the specification, in this case the input description. The software designer's perception is of unlimited resources; there is no concern about where a procedure is executed or how much of the total resources available it uses. Functional, rather than resource partitioning can thus be afforded by the software programmer.

Procedural inlining excludes recursion within the input specification. This limits the type of application that can be synthesised using CSiC . Only iterative specifications can be used as input. This style of coding is typical in this field of application. Most image processing algorithms tend to use 2 dimensional arrays of data and scan through them in an iterative manner [96].

Inlining simplifies the analysis task, reducing the complexity of the parser and providing an input description that more closely represents a hardware implementation than does a procedural hierarchy.

3.4 Parsing the Input Description

The front end to CSiC is an LR parser generated by Bison [98]: the GNU project parser generator. The parser utilises a lexical analyser generated by flex [99].

Both generators utilise the public domain C grammar provided by Jeff Lee of Gatech [100]. The grammar is based on the 1985 draft by the ANSI committee for the C language.

The parser has two main functions: to read and check the input description and to build structures that internally represent the same. The first of these tasks is an inherent function of LR generated parsers. Warnings of syntax errors in the description are supplied to the user. The error checking is aimed at common errors that might occur as a result of procedural inlining. Coded descriptions passed to CSiC should first compile properly using a C compiler.

3.4.1 The Statement Flow Graph

```
sum = 0;                        /* S1 */
for (i = 0;i < I;i++) {         /* S2;S3;S4 */
  sum += a[i+1][i];             /* S5 */
}
av = sum / (i - 1);             /* S6 */
```

Figure 3.2. Example Input Fragment

During the recursive descent parse a syntax tree is constructed. This is called the Statement Flow Graph (SFG) and is the basic representation used by all further stages of the compiler. It is essentially a control flow graph which supports iterative and conditional evaluation.

Each node in the SFG represents a program statement. Each statement node contains some form of expression represented as a binary tree structure. Nodes in these expression trees contain either operators or operands. An operand is typically either an identifier or a constant, an operator can be any of the C language operators.

Array expressions are represented as sub-trees. At the root of an access sub-tree is an index operator. This has two operands, an array reference and an index expression. The array reference can be either another index-rooted sub-tree or an identifier that gives the array name. This allows easy representation

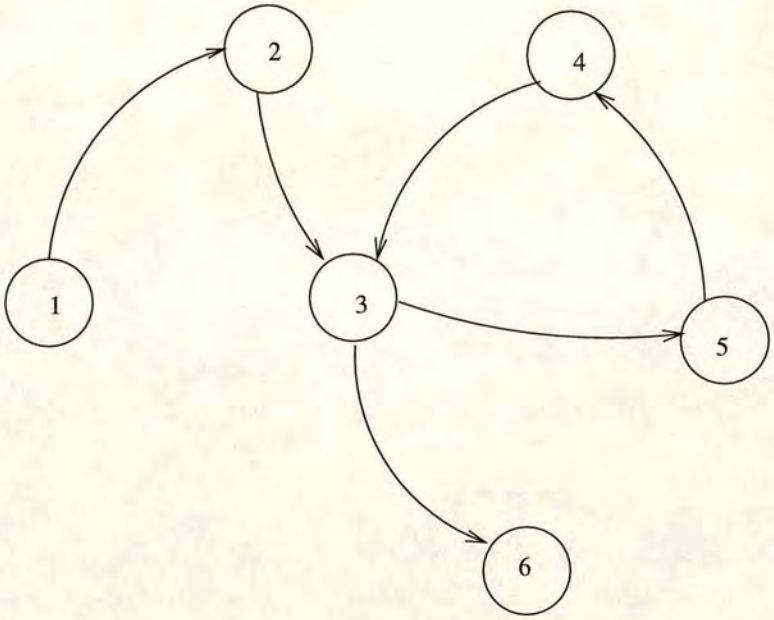

Figure 3.3. Statement Flow Graph for Input Fragment

of multidimensional arrays in the SFG.

Each identifier contains an entry in the symbol table with details relating to its size and type. Control is passed between statements as specified in the input description code. This flow is represented in the statement graph by edges between statement nodes. Edges can represent conditional or unconditional control flow. A node can be the source of at most one unconditional edge and up to two conditional edges. Statement hierarchy is represented by a mixture of conditional and unconditional edges between statement nodes.

Figure 3.2 contains an example fragment of code, shown as a SFG in Figure 3.3. The *for* loop is converted into a statement tree. The nodes in the tree represent the statements as shown. The conditional arc taken from node 3 depends upon the result of the comparison $i < I$. If the condition is true the True edge is fol-

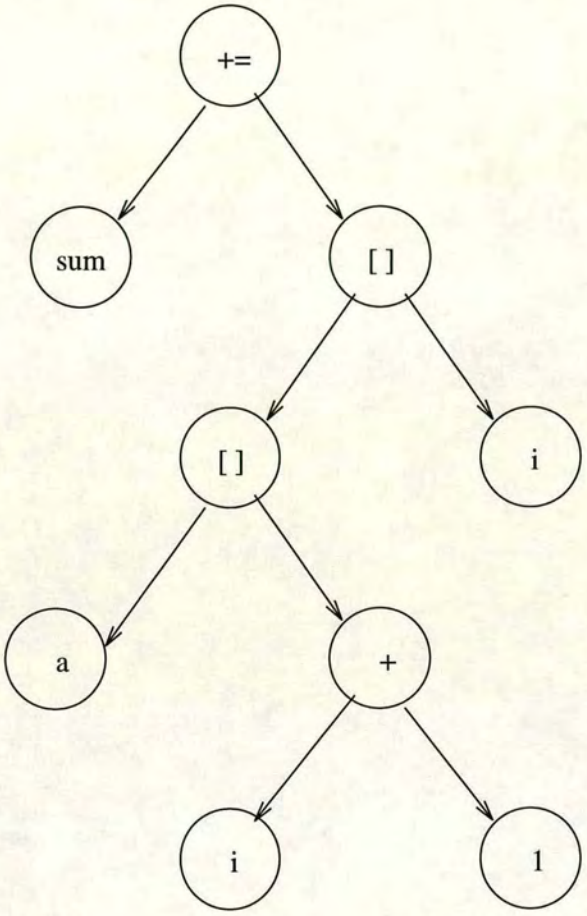

Figure 3.4. Statement Representation

lowed to statement 5. Otherwise the loop is terminated and execution continues with statement 6.

The internal structure of statement S5 in Figure 3.2 node is shown in Figure 3.4. Left and right edges emanate from each node in the tree, indicating the operands used by the operators. This illustrates the structure of array expressions in CSiC .

3.4.2 Normalising the Tree

Once the parse is complete and the statement graph for the input description is built, a number of standard transformations are applied to it. Expression

statements are reduced into triplet form. A triplet is an expression of the form

$$a = b \ \phi \ c$$

where a, b and c are simple operands, either variable or array references, and ϕ is an operator of the C language. In the case of a, b or c being an array reference CSiC reduces the statement to the form

$$a' = b'$$

with either a' or b' being an array expression, but not both. Additionally array index expressions are limited to a single identifier. This means that all array indices are calculated separately from the array expression itself in the normalised code. Figure 3.5 shows statement S5 from Figure 3.2 decomposed into triplet form.

```
_tmp1 = i + 1;
_tmp2 = a[_tmp1][i];
sum = sum + _tmp2;
```

Figure 3.5. Statement in Triplet Form

Reducing complex statements to this form is a standard compiler technique [52]. It simplifies analysis of the description and again, as in the case of inlining, it brings the description closer to the level of an eventual hardware implementation. In this case the ϕ operator would be some standard function performed by an ALU or other such hardware module. The triplet form also simplifies the "optimisation" of the description; common subexpression elimination [101] can be easily performed by searching a list of such statements and identifying results that have previously been calculated.

Such optimisations are not directly relevant to the main function of CSiC . It is primarily concerned with analysing array references within the input description, as these references represent the I/O interface between the processing core and the memory of the eventual hardware implementation.

However such analysis is itself aided by the statement graph being in the triplet form. Furthermore, future development of the compiler will require synthesis of the computational part, and for this such optimisations will be necessary.

3.4.3 Basic Blocks

```
leader_set = φ
∀ n ∈ statement_graph do
 if (indegree(n) == 0) /* picks start node /*
  leader_set = leader_set ∨ n
 fi
 ∀ e ∈ adj_edges(n) do
  if (is_goto(e)) /* conditional/unconditional edge */
   ∀ u ∈ adj_nodes(n) do
    leader_set = leader_set ∨ u
   od
  fi
 od
od
```

Figure 3.6. Find Basic Blocks Algorithm

A basic block is a piece of straight line code [51]. This implies a single point of entry and a single point of exit. Thus conditional edges between statement nodes represent transitions between basic blocks, as there will always be an alternative edge to a conditional transition.

All statements in the graph are labelled with their basic block number. The algorithm for labelling basic blocks is taken from [51] and is shown in Figure 3.6. This algorithm finds the basic block leaders, the set of statement nodes which are the first nodes in of basic blocks. All statement nodes are then labelled with the number of their preceding basic block leader.

CSiC builds a Global Flow Graph (GFG) using the block numbers. This graph gives the connections between different blocks and provides an invaluable resource for data-flow analysis applied in later stages. The GFG is constructed using the leader_set constructed in Figure 3.6. All the leaders in the SFG are

visited and a node is created for each of them in the GFG. These nodes represent the blocks in the GFG. The edges between blocks in the SFG are then replicated in the GFG, giving a graph which represents the flow between blocks.

3.4.4 Normalising the Variables

Two block level transforms are applied to the statement flow graph prior to subsequent analysis. These further the normalisation of the SFG, and extract static information from the input description. The two stages are constant propagation and multiple lifetime removal. These transforms operate on scalar variables. Both use data flow techniques found in [52] and [51].

Constant propagation involves discovering those variables that are assigned a constant value and replacing all references to those variables with the constant value, provided the path between each reference and the constant definition does not cover a further (non-constant) definition of the same variable. This improves later analysis as all static information implicit to the input description is extracted and made explicit in the internal representation used by CSiC . User input can contain symbolic information that improves the readability of algorithms, without affecting the synthesis process.

Multiple lifetime removal searches out scalar variables that are used and then redefined either once or several more times. This is a common programmer's technique, especially with loop induction variables. Multiple lifetimes lead to dependences existing between different sections of code, which would otherwise be independent if one section did not have to wait for a scalar variable to become free. This has ramifications for Process Scheduling[1].

Lifetime Analysis

Both transforms require lifetime analysis for the variables in the SFG. This data flow technique finds two sets of variables for each basic block: the *LIVE_IN* set, containing those variables "live" at the start of the block; and the *LIVE_OUT* set, containing those variables "live" at the end. A variable is said

[1]Section 4.1.1

```
∀ b ∈ G do LIVE_IN[b] = φ od
change = true
while change do
 change = false
 ∀ b ∈ G do
  LIVE_OUT[b] = φ
  ∀ n ∈ adj(b) do
   LIVE_OUT[b] = LIVE_OUT[b] + LIVE_IN[s]
  od
  oldin = LIVE_IN[b]
  LIVE_IN[b] = USE[b] + (OUT[b] - DEF[b])
  if oldin ≠ in[b] then change = true fi
 od
od
```

Figure 3.7. Algorithm for Finding $LIVE\_IN[]$ and $LIVE\_OUT[]$ Sets

to be "live" at a particular point in the program if it has been defined before, and will be referenced thereafter.

Example	# blocks	# itns	time(s) on 486x33Mhz PC
1	192	34	20.58
2	22	6	0.29
3	33	7	0.25
4	88	16	1.73

Table 3.1. Results Table for Algorithm of Figure 3.7

In order to find these sets two other sets are first constructed for each block, b in the SFG. These are $DEF[b]$: those variables defined before they are read in b; and $USE[b]$: those variables read in b before they are defined. From [52], the equations relating these to the solution sets $LIVE\_IN$ and $LIVE\_OUT$ are:-

$$LIVE\_IN[b] = USE[b] \cup (LIVE\_OUT[b] - DEF[b]) \quad (3.1)$$

$$LIVE\_OUT[b] = \bigcup_{s \in adj(b)} LIVE\_IN[s] \quad (3.2)$$

The operator in Equation 3.2 is the *confluence* operator. It performs a logical operation (in this case union) on the nodes indicated. Here the set $LIVEOUT[b]$ is assigned to the union of the $LIVE\_IN[]$ sets of the blocks adjacent to b[2].

These equations can be realised using a round robin algorithm which can be found in both [52] and [51]. An example of this algorithm is shown in Figure 3.7.

Table 3.1 shows the results of running the algorithm on a number of the test cases. It shows, as is claimed in [52], that the algorithm tends to iterate the outer loop a minimal number of times before reaching a stable state and thus is $O(N)$ where N is the number of block nodes.

Two of the examples in this and later tables are covered in detail in Chapter 6.

- Example 1 is an algorithm that uses an Efficient Hough transform for finding the eyes in an image of a face. Thanks to Ann Duncan for this example.
- Example 2 is taken from [27] and performs a comparison of fingerprint images.
- Examples 3 & 4 are parts of a Vector Quantization algorithm for image compression [25].

Reaching Definition Analysis

Constant propagation requires reaching definitions for the input description. A similar algorithm is used to obtain the $REACHED\_IN$ and $REACHED\_OUT$ sets for the basic blocks, as for the live variable sets. The equations governing the generation of these sets are

$$REACHED\_IN[b] = \bigcup_{p \in pred(b)} REACHED\_OUT[p] \quad (3.3)$$

$$REACHED\_OUT[b] = GEN[b] \cup (REACHED\_IN[b] - KILL[b]) \quad (3.4)$$

The confluence operator in this case is the union of the $REACHED\_OUT$ sets for the blocks that are predecessors to b. The GEN sets contain those variables

[2]the flow graph of blocks is directed

that are defined in each basic block. The $KILL$ sets contain all those definitions in the program that are killed by the generations in the block. The algorithm for generating the GEN and $KILL$ sets is $O(N)$, where N is the number of statements in the code.

The same round-robin type algorithm as was used for the live variable analysis is utilised for the reaching definition sets. It performs similarly here. Results for this and for the algorithm for generating the GEN and $KILL$ sets are given in Table 3.2.

		RDA		GKA
Example	# nodes	# itns	time	time
1	192	28	110.32	212.9
2	22	19	1.74	3.1
3	33	9	1.14	3.1
4	88	8	2.28	13.8

Table 3.2. Results for Reaching Definition and GEN and KILL Analysis

Constant Propagation

This transform makes use of the reached definition and live variable sets. All blocks are searched iteratively and the used variables are flagged. These are variables that are used before being defined in the block. Each used variables' reaching live definition is found and if this is a constant then the variable is replaced with the constant value. A reaching live definition is that definition of the variable that is also in the $LIVE\_IN$ set for the block.

Multiple Lifetime Removal

A further transformation stage is required for later analysis, the object of which is to remove multiple variable lifetimes within the program. If a variable is first initialised and then reinitialised at a later stage to a new value, this constitutes two lifetimes. The effect of this is to make the blocks succeeding the second initialisation appear dependent on the completion of the blocks in the first lifetime.

In order to remove this dependence all occurrences in the second lifetime are replaced by a new variable.

This stage uses live variable analysis [51] and [52], a standard data flow technique, covered above in Section 3.4.4. The algorithm used in performing the multiple variable substitution is given in Figure 3.8.

```
Find all births and deaths in the SFG for each variable
Find those variables with multiple births and at least one death, Vm
∀ v ∈ Vm do
  get the block lists for each birth-death pair, Lv
od
∀ list, l ∈ Lv do
  replace v in each block with a new variable
od
```

Figure 3.8. Algorithm for Multiple Variable Substitution

The births and deaths in the SFG are given by the two equations,

$$\forall \text{ blocks } b : births[v] = \bigcup_{v \in LIVE\_OUT[b] \cap v \notin LIVE\_IN[b]} b \tag{3.5}$$

$$\forall \text{ blocks } b : deaths[v] = \bigcup_{v \in LIVE\_OUT[b] \cap s \in succ(b) \cap v \notin LIVE\_IN[b]} (b, s) \tag{3.6}$$

Note that the set of deaths has to store two block numbers, indicating the edge leaving the block on which the variable ceases to live.

The block lists L_v are found by starting at a birth block for the variable and following the control flow until an edge is encountered that is contained in the death set for the variable. Each block passed is added to the list. Once the lists are complete then variable substitution is done using these block lists to traverse through the structure.

3.4.5 Other Preliminary Tasks

Much static information is contained in the original input description which is useful to succeeding stages of the compiler. This information is gathered from

the statement flow graph before embarking on further stages.

The statement flow graph can be represented as a basic block graph, with the nodes of the graph representing a block of straight line code. Some analysis is done using this graph, which provides information useful in later stages.

Sets of exclusive blocks are constructed. These comprise blocks on either branch of an if-then-else construct within the code. Thus for each block b, a set $excl(b)$ exists, which contains all the blocks in the exclusive branch, if one exists.

A block ordering is also taken which, when used in conjunction with the exclusive block sets gives a relative order of execution for the blocks in the input description. This block ordering is the so-called depth-first ordering of the block graph, an algorithm for which is given in [52].

Loops in the description are indicated by *back-edges* in the block graph. A back-edge has a target node which *dominates* its source node. A graph node is said to dominate another if every path leaving the first passes through the second (see below). The back-edge of a loop is between the exit node of the loop and the loop header node. Intuitively, the head node must dominate the tail.

Back-edges do not require detection through dominance analysis in CSiC, as these edges are flagged during the construction of the statement flow graph. The blocks contained in a loop are found by identifying the header and exit nodes of the loops via the back-edges and then counting in depth-first order from the header until the exit is reached. All the intervening blocks are enclosed by the loop.

The statement orders within blocks are also stored. The relative lexical order of a pair of statements in the description can now be established using the various sets. This information is required for dependence analysis within loops, see Section 3.5.1.

Finally, the block dominances are calculated. A block dominates another in the program flow, if all paths through the first block also pass via the second. This relation is invaluable in performing the inter-statement transforms, covered in Chapter 5. The algorithm used in CSiC for determining dominance relationships is taken from [52]. It is shown in Figure 3.9.

The first block node in the graph is only dominated by itself, so the algorithm initialises its set to itself. All other dominance sets are initialised to the set of

```
D(n_0) = n_0
∀n ∈ (N − n_0) do
  D(n) = N
od
change = true
while (change) do
  change = false
  ∀n ∈ (N − n_0) do
    D(n) = n ∪ ⋂_{p∈pred(n)} D(p)
  od
  if (change in a set) change = true
od
```

Figure 3.9. Algorithm for Determining Block Dominances

all nodes. The algorithm then iteratively reduces these sets according to the dominance sets of their predecesors.

Preprocessing Loops

During the parse the loops are numbered. Their nesting order is recorded as a set of indexed lists, each list containing the loops nested by the index loop. This structure allows easy reference to a statement's or block's nesting levels, in turn giving fast access to loop induction variables and their bounds.

Loops that exist on separate branches of if-then-else statements are also noted. These are exclusive loops: either completely exclusive as in the case of the if statement not being itself nested by an outer loop; or partially exclusive, where the if-then-else statement is surrounded by an outer loop.

The bounds of all loops are calculated where possible, those that are not statically realisable are marked as such. The iteration counts for the remaining loops are also stored.

3.4.6 Some Parsing Results

Table 3.3 contains the results for the four examples presented previously. This table shows the time spent (in cpu seconds) performing the various tasks presented in Section 3.4.

Ex	# lines	# AS[3]	# loops	Parse[4]	MT[5]	FBB[6]	CP[7]	MLR[8]
1	904	41	42	2.4	1.0	1.1	0.53	139
2	89	12	6	0.35	0.2	0.21	0.16	0.18
3	138	12	10	0.34	0.35	0.1	0.11	0.22
4	326	69	23	0.92	1.3	0.35	0.56	0.23

Table 3.3. Parsing and Preprocessing Results

The results in Table 3.3 summarise the times taken by CSiC for performing the tasks described above. The rather anomalous result for Example 1 for the Multiple Lifetime Removal Transform is a result of the large number of blocks in the code, caused by a high loop count (42) combined with a large number of conditional statements (27).

3.5 Basic Analysis - Block Level

On completion of the parse a statement graph exists which, together with the symbol table, is an internal representation of the input code. All subsequent processing in CSiC takes this statement graph and symbol table combination as a starting point. This means that the result of transformation of the statement graph can be fed back into the analysis stage in an iterative manner.

The basic analysis aims to provide data for the succeeding stages in order that they can synthesise hardware organisations effectively. Two types of basic

[3]Access Statements
[4]Includes SFG construction
[5]Making Triplets
[6]Finding Basic Blocks
[7]Constant Propagation
[8]Multiple Lifetime Removal

analysis occur: data flow analysis and control flow analysis. Both these processes use the SFG and basic block of the input description.

The next stage of the analysis is to determine the dependences within and between the basic blocks. Two different types of dependence are looked for: scalar dependences and array dependences. Array dependences are important for the synthesis of the memory hierarchy as it is taken as an initial assumption that array accesses refer to data held in main memory. The following sections outline dependence analysis as performed in CSiC .

3.5.1 Inter-Statement Dependence Analysis

An important part of the analysis of memory behaviour in CSiC relies on information about array accesses. In general these accesses take place within loops, and these give an initial ordering of reads and writes to array data. The syntactic ordering of array statements in the loop bodies shows how the algorithm operates on the data held in the arrays. It does not, however, give full information regarding the order of precedence of those memory accesses in order to preserve the semantic meaning of the algorithm. In order to obtain more of this information it is necessary to perform dependence analysis on the array statements in the loops.

```
for (i = 0;i < N;i++) {
  a[i] = ... /* S */
    ...
  ... = a[i+1] /* T */
}
```

Figure 3.10. Independent Statements

Using a reference to an array name, as in the case of a scalar, as proof of dependence is insufficient. Figure 3.10 shows this. Statement T is not dependent upon statement S as T will always read a location before it is written by S. S is dependent upon T however, as it writes locations read by T in the previous loop

iteration. This is an anti dependence between T and S, written T $\delta^*_<$ S. The direction of the dependence (<) indicates that it occurs for a later iteration of S than of T. The dependence has a distance of -1, and is said to be *carried* by the loop.

As the dependence is carried by the loop, S and T need not be executed in the order given in the code, with respect to the access to memory. As long as all other dependences are honoured, changing the order of S and T will not affect the original meaning of the code.

The topic of array dependence analysis was reviewed in Section 2.3. The present section details the techniques used by CSiC to perform this analysis.

```
for each pair of statements with the same array
 for each dimension of the array
  for each level of nesting
   for each direction vector
    if gcd test indicates dependence
     if banerjee test indicates dependence
      create appropriate dependence
```

Figure 3.11. Basic Method for Inter-Statement Dependence Analysis

Precedence for array access statements implies discovering if two instances of a statement access the same memory location. In the case of scalar variables this can easily be accomplished using the syntactic ordering of the statements and the identifiers used by the two statements. For array statements the identifier (array name) is not enough, as an index into the statement is also required, which is usually itself symbolic. Typically these indices are a function of the loop induction variables whose values are set by the expressions in the nesting loop statements.

Thus for a given array statement we want to find out whether any succeeding statements access the same location. In order to do this we have to find out if the array index expressions evaluate to the same value. A method for accomplishing this is reviewed in Section 2.3.1.

Before testing two statements for dependence their relative positions in the

code must be established. Statement ordering, performed as one of the preliminary tasks after the input parse, provides a means of ascertaining the relative order of two statements. A number of possibilities exist for identifying a succeeding statement. These are:-

1. A syntactically later statement in the same iteration.
2. An exclusive statement in the same iteration.
3. The same statement in a later iteration.
4. A syntactically greater statement in a later iteration.
5. A syntactically lesser statement in a later iteration.
6. A syntactically greater statement in a different loop nesting.

CSiC attempts to test all the above cases within the input description. The type of dependence found, if any, will depend on whether the two statements read or write to the array. For deeply nested statements (ie with more than one surrounding loop) the type of dependence found will also depend upon which iteration comes first, the source or the sink of the dependence.

In Section 2.3.1 methods for performing such tests were introduced, namely the gcd and banerjee tests [78]. Conservatively, dependence is always assumed, unless the tests show it does not exist. CSiC uses these tests on all array access statements in the program description. Section 2.3.1 gives an algorithm that can be used to find the absence of dependence between two statements contained in nested loops.

This algorithm (Figure 2.3.1) forms the basis of the array dependence testing in CSiC . All array statements are compared with each other in order to get a full picture of the program's dependence relationships. Tests must be performed for each array dimension and at each level of loop nesting. This leads to a large number of tests for each statement pair. The amount of testing can be reduced by weeding out tests which will not lead to useful results. The basis of the method used here is shown in Figure 3.11.

Before carrying out this testing some preparation must be done. As well as the relative statement positioning discussed above, the diophantine equations must be extracted from the array statements and the nesting loops identified. The latter information is gathered from the loop numbering added during parsing along with the loop nesting hierarchy built after the parse.

In order to obtain the diophantine equations for the array indexes it is necessary to have lists of the induction variables used in all loops. The diophantine equations must be in terms of the nesting loop induction variables.

Each array index expression is examined for its constituent variables, or index variables. If any of these are not induction variables or constants, then it must be discovered whether they are linear functions of the induction variables.

This is achieved by backtracking from the array statement and iteratively building an expression tree by expanding each unknown by its definition. This is continued until the expression tree contains only constants and induction variables, or until a unique definition for a variable cannot be found.

Reaching definition analysis comes in useful again here. The point at which the index variables are defined is given by the reaching definition set for that block[9]. This will eventually lead to their definitions in terms of induction variables or indicate that no such relation exists. In the latter case dependences between the statement and others must therefore be assumed as dependence can not be disproved.

In the case of the index variables being wholly defined by enclosing induction variables and constants then the diophantine equation is constructed, and the coefficients of the induction variables stored for each statement. The method for dependence testing requires that index expressions are linear functions of their induction variables. If functions are non-linear then dependence must be assumed.

The loop bounds are also held for each statement as these are also required by the testing procedures. The algorithm described in Section 2.3.1 is implemented within CSiC ; this procedure combines with the algorithm in Figure 3.11, taking the above information and checking for dependences.

---

[9]See Section 3.4.4

Ex	# Array Stmts	Max Loop Depth	Max Array Dim	Time(s)
1	41	4	3	110
2	12	6	3	100
3	12	5	3	7.9
4	69	3	3	10

Table 3.4. Times for Dependence Tests

The efficiency of the dependence testing is very sensitive to the number of array statements being examined and their level of nesting, as can be seen by an examination of the algorithm in Figure 3.11. Some improvement of the performance is obtained by checking both statements for any index expressions non-linear in the induction variables. These statements are not compared with others.

The most important checking avoids performing unnecessary dependence tests by careful selection of direction vectors presented to the test for each statement, as outlined in [78]. This is accomplished by giving the first dependence test a direction vector composed entirely of '*' symbols. These are "don't care" directions, so this first test looks to see if any dependences exist at all.

Each successive test incrementally builds up the direction vector, and tests in the same way. The second direction vector tried is (<,*,...,*). If dependence is proven for this vector then we proceed with (<,<,*,...,*), otherwise the next is (=,*,...,*). If this fails then (>,*,...,*) is tried. Testing continues in this way until complete direction vectors are found for dependences that exist. This method performs a pruned search of the direction "space", and improves the performance of the testing.

The times for dependence testing of the four example applications are shown in Table 3.4.

Once this is complete some filtering is performed on the dependences found. For inter-statement dependence analysis on multi-dimensional arrays contained in multiple nested loops, a dependence exists between statements only if the same dependence is found for each array dimension. Another filter applied is to remove all dependences that do not have a forward direction for S δ T. This mean that any dependences found between S and T where an S iteration occurs after a T

iteration is removed. Any dependence with a direction vector whose leftmost non '=' symbol is '>' falls into this category. This type of dependence indicates that T is executed before S. As T depends upon S from the dependence expression, this cannot be the case.

All such filtered dependences are kept for further analysis but are not referred to in the rest of this chapter.

The dependences found are stored in graphs and used by later stages of the analysis. The Block Access Graphs (BAGs), discussed in Section 3.5.3, are updated from these dependences. The BAGs reflect the order of array accesses that maintain the meaning of the original description. By updating the edges in these graphs with the results of the array dependence analysis, the semantic meaning is still maintained but false relationships based on treating the arrays as scalar quantities is removed. The code fragment in Figure 3.10 would originally have forced the access in statement T to have come after that of statement S. After updating the dependence is removed showing that the accesses can proceed in any order.

3.5.2 Intra Block Scalar Analysis

Scalar dependence information is important because it defines the semantic behaviour of the program that operates behind the I/O interface. Additionally scalar flow is important for calculating array indices within the code. A combination of both sets of dependences will be used in ascertaining the order of execution of basic blocks within the description.

The first step taken is to find the dependences between statements within each block. This is accomplished by building a graph of the nodes within a block and then adding edges to the graph. These indicate the order of execution of the statements. A graph is produced for each basic block in the statement graph, these are called Block Dependence Graphs (BDGs).

This process removes the strict order defined by the sequence of statements in the code description. The dependence graph shows the ordering of statements within each block necessary to maintain the semantic behaviour of the program

```
∀b ∈ basic_blocks do
 tar = last_block_stmt(b);src = prev_stmt(tar)
 make load and store sets for tar
 while (src ∈ basic_block(b)) do
  make load and store variable sets for src
  ∀v ∈ load_vars(tar) do
   if (v ∈ store_vars(src)) then
    new_edge(src, tar);remove v from load_vars(tar)
   fi
  od
  ∀v ∈ store_vars(tar) do
   if (v ∈ store_vars(src)) then
    remove v from store_vars(tar)
   else if (v ∈ load_vars(src))
    new_edge(src, tar)
   fi
  od
  if (store_vars(tar) == φ ∧ load_vars(tar) == φ) then
   tar = prev_stmt(tar);src = prev_stmt(tar)
   make load and store sets for tar
  else
   src = prev_stmt(src)
  fi
 od
od
```

Figure 3.12. Algorithm to extract Block Dependence Graphs (BDGs)

specified in the input description. It also exposes the potential parallelism available in executing the statements in the statement graph, at the basic block level. The algorithm used to generate the BDGs in CSiC is shown in Figure 3.12.

The algorithm operates on the statement nodes within each block. It finds dependences between statements by searching out flow dependences between scalar variables in statements. It also gives precedence to those statements that must use a variable before its value is overwritten.

Array accesses are treated as scalars at this stage. The next section details analysis of the dependences between the array accesses in the blocks. The results

of this analysis is used to update the BDGs by filling in the dependence direction and, where known, the distance for array accesses.

3.5.3 Intra-Block Access Analysis

The primary function of CSiC is to analyse access to array data in the program description in order to ascertain the I/O requirements of the algorithm. At this stage all arrays are treated the same, their position in the implementation's memory hierarchy is not yet known. In order to simplify the analysis a further set of graphs are constructed which show the dependence of array accesses relative to one another in each basic block. These graphs are called Block Access Dependence Graphs (BAGs). They are constructed by reducing the BDGs so that non-array access nodes are removed, with their adjacent edges joined.

```
_tmp14 = a[i];            /* S1 */
_tmp15 = _tmp14+1;        /* S2 */
a[i] = _tmp15;            /* S3 */
_tmp16 = b[i];            /* S4 */
_tmp17 = _tmp16+_tmp15;   /* S5 */
b[i] = _tmp17;            /* S6 */
c[i] = 0;                 /* S7 */
d[j][k] = i;              /* S8 */
```

Figure 3.13. Basic Block Code for BAG Example

Dependences between array nodes are updated using the results of the array dependence analysis covered in Section 3.5.1. This process ensures that the order of array accesses relative to one another is semantically consistent, and preserves the order of scalar flow beneath the access graphs.

Figure 3.14 shows the generation of a BAG from its BDG, the basic block code for the BAG is shown in Figure 3.13. Weights are added to the edges in the BAGs. These weights indicate the number of operations performed between each array access. This gives a crude measure of the immediacy of use of the array

data, and is used by the scheduler for conflict analysis.

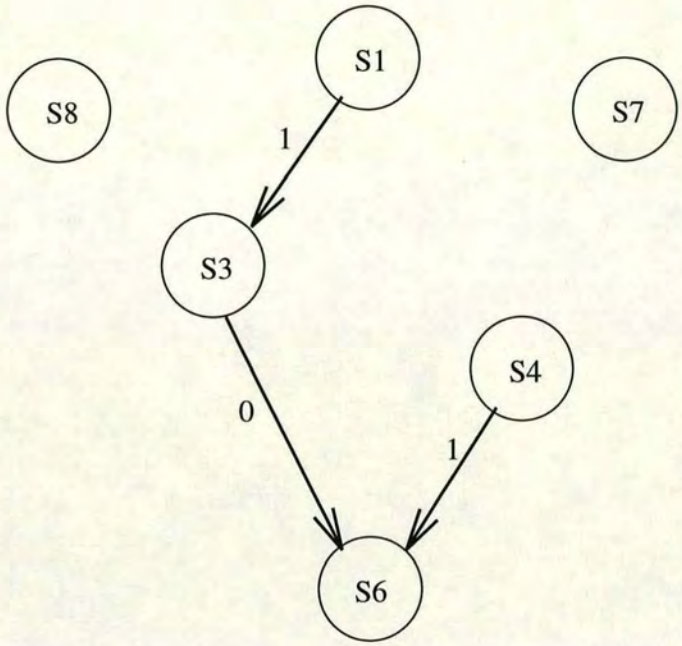

Figure 3.14. A Block Access Graph from a Block Dependence Graph

3.6 Summary

This chapter has introduced the CSiC tool, and some of the issues it addresses. The basic analysis performed on the input description is outlined, this will be used in the following chapters for the further synthesis and transformation tasks that the tool implements.

This basic analysis is common to many synthesis systems, the software compiler-like operations such as constant propagation and multiple lifetime removal are used in other synthesis tools. Much of this basic analysis is intra-block, which is a mature and well understood method for both hardware and software compilation. The array dependence analysis performed is largely still only used in software compilers for parallel machines [102], although it is beginning to penetrate the general compiler community [85]. There are no explicit references to it in the synthesis literature, the later CATHEDRAL tools [18], [37], [38] and

PHIDEO[74] tools recognise delays in synthesising memory architectures, but do not make clear how this is accomplished.

Further work at this stage would involve the improvement of the intra-block analysis. Trickey [65] makes the observation that intra-block parallelism in software code typically has a parallelism degree of two or three. He increases this by employing various block merging transforms[10] which increases the parallelism degree to between five and ten. Renaming of variables [103] provides another useful technique for increasing the degree of parallelism, this would also be a useful addition to the CSiC tool.

However in its current state, CSiC is mainly concerned with the relationships between different array statements, and this occurs both within and outwith basic blocks. As will be shown in Chapter 5, the transforms utilised by the tool are intrinsically inter-block in nature.

[10]See Section 2.1.4

Chapter 4

Synthesising the Hardware

This chapter describes the synthesis tasks performed by the CSiC tool. The previous chapter gave an introduction to the tool and detailed the preprocessing and basic analysis tasks performed by it on the input description. These included basic block and dependence analysis, for both scalar and vector variables. The various data structures built during these phases will be put to use in the stages described here.

The synthesis processes described here are concerned with the extraction of data from the input description, and the generation, from this data, of memory architectures. These processes include scheduling, allocation and evaluation. Transformation procedures are covered in the next chapter. They attempt to improve the success of the extraction procedures based on an assessment of the solutions produced. After application of the transformations some of the earlier analysis has to be performed again, which involves most of the analysis procedures detailed in Section 3.5.

The allocation performed is detailed in the section following that on scheduling. Once these tasks are completed a number of hardware organisations can be produced. Each of these has to be evaluated. This process is detailed in the third section below.

The steps required to produce hardware descriptions, in this case memory organisations, are common to many systems described in the literature. The task of scheduling instructions and processes and the subsequent allocation of

hardware to these has been the subject of much research activity [28, 16, 104, 105, 106]. With respect to the application of such methods to memory synthesis a relatively smaller number of publications are available. Notable amongst these are [18] and [74]. A discussion of the techniques utilised in these and other systems can be found in Section 2.2.

The technique presented here is multistage, having scheduling interleaved with a "logical" allocation phase. In this sense the technique is interdependent, as classified by Paulin and Knight [16]. It is applied to both process (ie nested loop) and block schedules. The parallelism in each type of schedule is used in the allocation of arrays to memory. The overview below gives a brief description of the method as it is applied to basic blocks. The two different types of schedule generation are then discussed in more detail in the sections below.

An optimal schedule is first produced for each of the basic blocks in the description. This is similar to the technique used in percolation based synthesis [61] where an optimal schedule is produced for a loop by successive unrolling of the loop body and percolation of operations towards the start of the block, as far as dependences in the code will allow. A somewhat different method is presented here.

This method finds an optimally parallel schedule for each block, according to the dependences in the loop code. This is analogous to, but weaker (at this stage), than the percolation method[61]. These "optimal" schedules are then used to produce global weighting matrices for the arrays in the input. These matrices are used for grouping arrays into memory units. Different architectures will have differing numbers of separate memory units, the matrices allocate arrays to these different blocks so as to obtain the maximum throughput for the algorithm. I call this "logical allocation". As a schedule is produced for each block in the description, there is a local influence on the global allocation strategy.

Final schedules for the basic blocks are then generated for each architecture, based upon the resources made available during the logical allocation phase. Each architecture is then allocated physical resources, for the logical memory units (LMUs) specified and also for any other components inserted into the memory hierarchy.

The process scheduling occurs before the logical allocation phase. Basic blocks

are treated as if they were nodes in their own dependence graphs and the arrays accessed within them are used to construct tables similar to the global access matrices described above. The figures from these are then added into the basic blocks' access matrices so that the global scheduling requirements have an effect upon the logical allocation.

Once these tasks have been performed it remains to evaluate the probable performance of the hardware so that both the user and subsequent iterations of the synthesis process can decide which solution to choose. This is detailed in Section 4.2 below. Due to different processes schedules, a number of different organisations are generated for each architecture. Each of these is evaluated separately.

4.1 Scheduling of Memory Accesses

The scheduling performed by CSiC aims to produce an efficient access profile for the algorithm's data, giving a number of different memory organisations. When constructing a memory architecture, a number of different factors to have to be considered. In CSiC these are,

1. The number of physical blocks of memory
2. What data is stored in which blocks
3. Sizes of the blocks
4. The type of memory in the blocks
5. The location of the blocks, eg on- or off-chip
6. The connectivity of the blocks

The tool addresses these factors with the intention of optimising the hardware synthesized so that it is able to meet the access demands of the algorithm. If these points are met effectively, then the synthesized hardware will be more likely to meet the required execution time for the algorithm.

In order to produce a schedule for the entire algorithmic description, scheduling needs to be performed at two levels, the intra-block and inter-block levels. These two processes use the data structures described in the previous chapter. Specifically the Block Access Graphs (BAGS) are used for intra-block scheduling and the Global Flow Graphs (GFGs) for inter-block scheduling.

Scheduling and allocation are closely related; the former requires some knowledge of what resources are available; whilst allocation of resources requires some prescience of the sequence of events that will occur in order to support them.

In the case of memory allocation a limited number of options exist, and for the scheduler these are mostly concerned with the amount of data that can be accessed in each time step. This depends upon the number of different sources of data available at each step, related itself to the number of different memory sources that exist in the system. As the CSiC scheduler is primarily concerned with the scheduling of memory accesses between processing core and memory, a different schedule must be constructed for each different memory organisation.

However, in order to do this the scheduler must have some information about the resources that exist in these different memory organisations. This is achieved by first finding an *optimal* schedule for the algorithm. Here an optimal schedule exploits all the inherent memory access parallelism in the algorithm. This is then used as a basis for generating memory organisations for which individual schedules are produced. The intention is that the solutions produced by this process are as close to optimal as possible. Optimal in terms of memory architecture has been taken to mean that all data is available as soon as it is required.

The next sections describe the process by which schedules and logical memory organisations are produced. Process scheduling is covered first. This is the first stage in generating the different architectures, one for each separate process schedule. Basic block scheduling is then covered, which leads to the "logical allocation" stage, followed by the full scheduling for each logical architecture produced.

4.1.1 Process Scheduling

A procedural program is typically broken down into tasks performed in a serial manner. The effect of inlining[1] such a program is to produce a sequence of "processes" that operate on arrays of data within the program. By process I mean a number of tightly nested loops surrounding sequences of statements that might include further processes. This gives a hierarchy of processes with layers of child processes existing at the same level.

The precedence of these processes is primarily governed by the procedural order coded by the programmer. Analysis of the dependence relationships between processes will lead to a different order of precedence, dictated by the data flow requirements of the processes themselves. An efficient hardware implementation of the algorithm will have to address itself to these precedences, just as a design engineer would have to.

Analysis of the dependences between basic blocks in the program code provides these precedences and allows the scheduling of processes relative to one another. An initial flow graph for each level can be built up by looking at the arrays that are accessed by each process and the types of accesses performed. Processes are classed as consumers and producers of particular arrays and dependence relationships are inserted into the flow graph. Some processes will also have scalar relationships, where some function is performed on an array in order to produce a scalar result, which is then used as a "parameter" to another process. One example of such a situation is histogram equalisation of an image frame. Such relationships are also annotated into the process flow graph.

Making The Process Flow Graph

The process flow graph (PFG) is constructed from the global flow graph (GFG, see Section 3.4.3). Nodes in the GFG represent basic blocks in the code, and their connecting edges represent the flow of control between these blocks. Analysis proceeds in a hierarchical manner, and start nodes for loops and conditional

[1] See Section 3.3.2

```
if (x == 5) {
  for (i = 0;i < N;i++) {
    ...
      a[i] = ...
      ...
  }
  for (i = 0;i < N;i++) {
    ...
  }
}
else {
  for (i = 0;i < N;i++) {
    ...
  }
}
```

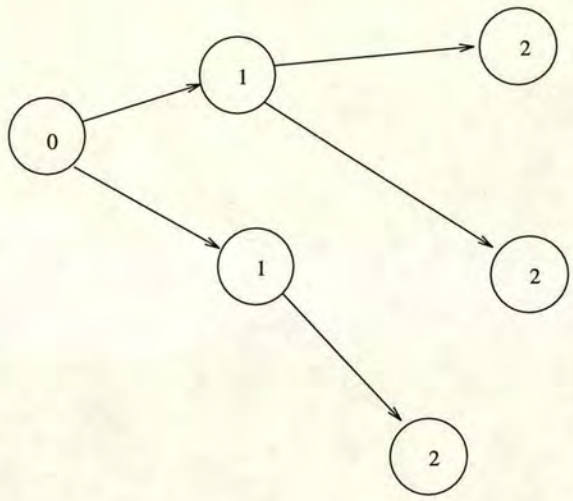

Figure 4.1. A Code Fragment And Its PFG

statements are treated as super-nodes for their connecting sub-graphs. This produces a layered graph composed of sequences of super-nodes, each representing a complex C statement (either a loop or a conditional) in the input description. See Figure 4.1 for an illustration of this.

The connecting edges between these super-nodes are unconditional. The PFG is essentially a copy of this structure but with the control flow edges between super-nodes replaced by the dependence relationships between the super-nodes. This involves some simple analysis of the data used by the sub-graphs contained in the super-nodes.

The GFG is traversed in a depth first fashion, and variable sets are made for each super-node. These sets contain variables written and read within the sub-graphs. The intersections of these sets show the types of dependences that exist between super-nodes, and therefore the precedence that each node has over the others.

The multiple-lifetime removal transform described in Section 3.4.4 will have removed any variable re-use that occurs in the input description. A typical instance of this situation is the re-use of loop induction variables. This creates

an unnecessary dependence between adjacent loops which would show up in the PFG as an edge between super-nodes. In the absence of any other dependences between the two nodes this would result in the loss of an opportunity to exploit parallelism between the two nodes.

Creating the Process Schedules

Once the Process Flow Graph is complete an initial schedule can be constructed for its execution. This proceeds in much the same way as for access scheduling in the basic blocks (see Section 4.1.2 next), and a maximally parallel schedule is sought.

The PFG is traversed hierarchically, and at each level a maximal schedule is constructed for the super-nodes there. The scheduling follows a ASAP method, using the dependences in the PFG to determine the ordering of super-nodes in the schedule list. Once the entire graph has been traversed a hierarchical ordering of nodes is realised.

Control "steps" in process schedules are elastic; for block scheduling they are fixed to the basic time unit, a memory access. Process schedule steps contain sequences of accesses that are governed by the bounds on the loops within each process. This allows for short connected processes to be parallelised with longer processes, if dependence relationships permit. No timing is performed at this stage however, only the scheduling of the maximum number of processes into a single step that dependence allows.

4.1.2 Generating the Optimal Schedule

In order to perform the logical data allocation, an optimal schedule must be produced for the internal description of the input. This involves scheduling each block with unbounded resources - maximally parallel scheduling. In this method it is assumed that all data required by statements is available at the earliest possible time, unconstrained by resources.

In terms of memory access this means that all array expressions in a given list of statements can be executed as soon as possibly allowed by the dependence restrictions imposed by the semantics of the code. In order to achieve this, each

basic block in the algorithm is taken in turn and its array statements maximally scheduled, as outlined below.

Maximal Parallel Block Scheduling

A Block Access Graph (BAG) exists for each basic block[2]. Each BAG contains one or more clusters of interconnected nodes that represent memory (array) accesses. Nodes within the clusters are connected via edges which indicate dependences between accesses. Two clusters therefore contain memory accesses that are independent of each other. These accesses can potentially be performed in parallel with each other, thus increasing the I/O throughput of the hardware implementation. Additionally there will be accesses within clusters that can also be performed in parallel; those that access different arrays.

The scheduling process aims to produce a schedule that achieves all of this parallelism, giving a maximally parallel schedule. This of course assumes unlimited resources. That is, an access occurs as soon as it is possible to do so. This simplifies the scheduling at the pre- allocation stage, leaving later stages to adjust the schedule according to the actual resources that have been allocated.

In order to schedule a block, each cluster in the block is first scheduled separately. The individual schedules are then merged to give a final schedule for the block. A prioritised list scheduling algorithm is presented below. This is used in CSiC for scheduling individual clusters.

Cluster Scheduling A variant of prioritised list scheduling is employed to place accesses into control steps. A control step is taken to be enough time to perform either a read or write operation to a single port memory. A virtual machine is assumed, composed of a number of independently accessible memory blocks, each of which contains one of the arrays declared in the input description. Thus for N arrays in the description, the virtual machine is composed of N memory blocks and has the ability to perform N memory accesses in parallel. The algorithm is shown in Figure 4.2.

Source nodes (those with no incident edges) are chosen as the initial accesses

[2]See Section 3.5.2

```
schedule_set = ⋃_{indeg(n)==0}  ∀ n ∈ cluster_nodes
step_counter = 0
while schedule_set != φ do
  Examine schedule_set for conflicts
  Resolve conflicts by moving some nodes to defer_set
  ∀ n ∈ schedule_set do
    schedule[n] = step_counter
  next_step_set = ⋃_{n∈schedule_set} next(n)
  ∀ n ∈ defer_set do
    if ∃ s ∈ next(n) ∧ s ∈ next_step_set
      next_step_set.del(s) fi
  schedule_set = defer_set ∪ next_step_set
  defer_set = φ, step_counter++
od
```

Figure 4.2. Cluster Scheduling Algorithm

to place into the schedule. This ensures that all branches in each cluster are picked up by the scheduler. The conflict analysis then decides which branches are to be executed first, where necessary.

Conflict Analysis Conflicts occur when two or more accesses in the schedule_set refer to the same array. These conflicts are logged to provide feedback information to the transform stage. Such conflicts are resolved by attempting to minimise the amount of register storage needed to hold the data.

Edges in the BAGs contain weights that give an indication of the longevity of data in the nodes. The node with the most immediately required data is kept in the schedule set, the other nodes involved in the conflict are removed and placed into the defer set for scheduling in later control steps. The longevity measure is computed during the basic analysis phase and is determined by the distance between source and sink array accesses in the block dependence graphs (BDGs[3]).

The current value of the step counter is assigned to the remaining nodes in the

[3]see Section 3.5.2

schedule set after conflict analysis is completed. The successors to these nodes are then placed into next_step_set. This set is then checked against the successors of those nodes that have been deferred (last loop in Figure 4.2). The successors to nodes are thus not scheduled until all predecessor nodes have been assigned a control step, maintaining the semantic meaning of the block's code.

```
1. set base_schedule = longest schedule from cluster_schedules
2. set insert_schedule = next longest schedule from cluster_schedules
  if no more schedules to insert then stop
  set base_step = 0;set insert_step = 0
  current_set = insert_set[insert_step]
3. ∀ a ∈ current_set do
    if a ∈ base_schedule[base_step]
      insert a into defer_set;delete a from current_set
    fi
  od
  ∀ a ∈ current_set do
    insert a into base_schedule[base_step]
    delete a from current_set
  od
  if defer_set == φ
    insert_step += 1;base_step += 1
    current_set = insert_schedule[insert_step]
  else
    base_step_ += 1;current_set = defer_set
    defer_set = φ
  fi
  if current_set == φ goto 2.
  else goto 3.
```

Figure 4.3. Merging Cluster Schedules

Cluster Schedule Merging In order to obtain a complete schedule for the basic block it is necessary to merge the cluster schedules. Whilst nodes in separate clusters have no dependences, there may exist nodes in the same control step that

access the same array. This would lead to a conflict in the final schedule for the block.

The method used to merge the cluster schedules is outlined in Figure 4.3. This method follows an As Soon As Possible (ASAP) strategy for placement of accesses within a particular schedule. The prioritising of clusters is based on the length of the critical path of each cluster. The longest critical paths are merged first with the intention of minimising the final schedule length.

```
_t1 = a[i];     /* S1 */
_t2 = b[i];     /* S2 */
_t3 = a[i-1];   /* S3 */
_t4 = _t2*_t3; /* S4 */
j = _t1+_t4;    /* S5 */
_t5 = a[i+1];   /* S6 */
b[i] = j*_t5;   /* S7 */
_t6 = d[i-1];   /* S8 */
_t7 = c[i];     /* S9 */
d[i] = _t6+_t7;/* S10 */
_t8 = c[i];     /* S11 */
_t9 = d[i];     /* S12 */
c[i] = _t8*_t9;/* S13 */
```

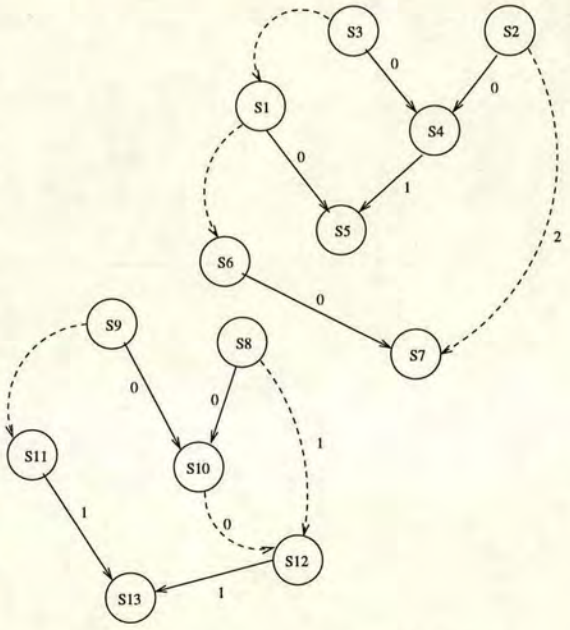

Figure 4.4. Basic Block Code Fragment and Its BAG

Example Schedule The following example illustrates the maximal scheduling. Figure 4.4 shows a basic block code fragment and its BAG, the weights on the BAG edges are shown. The BAG contains two clusters as separate sub-graphs. Figure 4.5 shows the schedules for the two clusters contained in the BAG and Figure 4.6 shows the final merged schedule for the block's code.

The resultant block schedules are optimal and exploit all possible parallelism within each iteration. Some schedules could be made more parallel at this point by wrapping the ends of the schedules around. This is only possible for tightly

1. load_a[i], load_b[i]	1. load_d[i-1], load_c[i]
2. load_a[i-1]	2. store_d[i], load_c[i]
3. load_a[i+1]	3. load_d[i]
4. store_b[i]	4. store_c[i]

Figure 4.5. Schedules for Clusters in Figure 4.4

1. load_d[], load_c[], load_a[], load_b[]
2. store_d[], load_c[], load_a[]
3. load_d[], load_a[]
4. store_c[], store_b[]

Figure 4.6. Final Merged Schedule for the Basic Block

nested blocks, ie those that comprise an entire loop body, and between loop begin and end blocks. In order to keep the scheduler simple this technique is utilised at the process schedule level, see Section 4.1.1.

4.1.3 Building The Block Allocation Matrices

The allocation matrices reflect the optimal allocation of arrays into memories with respect to each other. This is achieved by weighting the arrays against each other, the weight being based on the number of accesses made to each array in the weighting pair.

After each block has been maximally scheduled two sets of weights are made for it: the series weights and the parallel weights. These sets reflect the interactions between the different arrays in the block. The maximally parallel schedule for a block is represented as a list of lists of array accesses. See Figure 4.6 for an example. Each row in the list represents a control step and all arrays in each step are accessed at the same time.

As can be seen from this schedule the optimal allocation of arrays to different

Array	a	b	c	d
a	0	n	$2n$	$3n$
b	n	0	$2n$	n
c	$2n$	$2n$	0	$2n$
d	$3n$	n	$2n$	0

Figure 4.7. Parallel Allocation Weights

Array	a	b	c	d
a	$2n$	n	$2n$	$2n$
b	$2n$	0	n	$2n$
c	$2n$	n	n	$3n$
d	$2n$	$2n$	$3n$	$2n$

Figure 4.8. Series Allocation Weights

memories for the basic block in Figure 4.4 would be to have each array occupying a separate memory unit. This would allow the maximum throughput for the block, minimising the access bottleneck. Such an allocation is likely to be prohibitively expensive, and would also lead to an under-utilisation of the memory units. so some compromise solution is required. The serial and parallel weights provide the means for achieving this compromise.

Both sets of weights are obtained from the maximal schedule for each block. The weights are essentially a measure of the advantage of having two arrays in separate memory units. In Tables 4.7 and 4.8 n is the number of times the basic block is executed. It can be seen from the schedule in Figure 4.6 that arrays a and d appear in the same control step 3 times, hence their mutual weight of $3n$ in the parallel weight table.

The serial weights are also collected. These show the incidence of one array awaiting an access to another, enforced by the program semantics. The serial weight between accesses is analogous to the advantage of having the two arrays in the same memory unit. The serial weights are used to guide the allocation process, providing an arbiter for clashes in the parallel table, see Section 4.1.5.

4.1.4 Producing Weight Matrices for each Schedule

Each process schedule represents a different potential architecture for the final solution. A differing arrangement of relative process executions will place differing demands on the hardware. This has to be represented in some way during the logical allocation phase.

The Block Allocation Matrices (BAMs), described in the previous section, in association with the process schedules themselves provide this feedback to the allocation phase. An overall set of weights are derived from these structures and these are used in allocating the data arrays to memory.

The use of these two structures provides both a local and global level influence on the allocation process. Local information comes from the accumulated weights of the BAMs. Global information is obtained from the process schedule itself, guiding how the information from the BAMs is combined, and also providing some additional input itself.

The Global Allocation Matrices (GAMs) are produced by a summation of the BAMs across the SFG. This summation is guided by the precedences in the PFG. As with the BAMs two sets of matrices are constructed:

$$\text{pgam(a,b)} = \sum_{k}^{pfg} \text{pbam(a,b)} + \text{parw(a,b)} \tag{4.1}$$

$$\text{sgam(a,b)} = \sum_{k}^{pfg} \text{sbam(a,b)} \tag{4.2}$$

The summation over the PFG performed by the first term of the right hand side of both equations 4.1 and 4.2 has to take into account processes that are scheduled in parallel. A bottom up traversal of the PFG takes place. Those super-nodes that are executed in parallel have their summations compared for each pair of arrays and the minimum weight value is taken as the result.

The second term in Equation 4.1 allows for arrays in parallel super-nodes that would benefit from separate locations. This information is unavailable in the BAMs as they only provide information for arrays accessed in the same block. This second term gives global feedback for the allocation process.

Once the GAMs have been produced for each schedule the logical allocation can proceed, followed by scheduling proper for each schedule.

4.1.5 Logical Allocation

The weights contained in the GAMs give relative measures of the access load between different arrays for the entire algorithm. The Parallel GAM shows the parallel access gain obtained for placing two arrays in separate memories. The serial GAM shows the number of sequential accesses that have to be made to two arrays, as dictated by the dependence relationships.

The logical allocation process uses these weights to decide the arrangement of the arrays into multiple memory units. The GAM elements are first prioritised in order of decreasing weight. A number of LMU are then chosen, usually this is set by just iterating from 2 up to the maximum number of arrays in the input description.

The algorithm used for logical allocation of a set of arrays onto some number of separate memory units is given in Figure 4.9. In this algorithm an array of program array identifiers is created, allocated[], which at the algorithm's end will contain the number of the LMU that the array has been placed into. The algorithm has three main parts after initialisation: parallel allocation, serial allocation, size allocation.

The two procedures in Figure 4.9, choose_one, choose_best are used in both of the first two parts. The third procedure, allocate_rest, allocates those arrays not placed by the first two. The selections are made dynamically so that decisions are based on what has been allocated before, rather than by statically setting the order in the priority list.

Parallel Allocation

The first pass in Figure 4.9 attempts to allocate all arrays based on their parallel access profile. The parallel allocation matrix gives relative measures for the benefit obtained in having pairs of arrays in separate memories. A list of these pairs is constructed and arranged in decreasing order of parallel weight, from the matrix. "Quicksort" routines attached to the LEDA list class ensures this is done

```
∀ a ∈ arrays do
  allocated[a] = 0
od
make parallel priority queue
while priority_queue != φ do
  top_pair = priority_queue.pop()
  while !allocated[top1] ∨ !allocated[top2] do
    array = choose_one(top_pair)
    lmu = choose_best(array)
    allocated[array] = lmu
  od
od
if ∃ a : allocated[a] == 0
  make serial priority queue
  while priority_queue != φ do
    array = priority_queue.pop()
    if !allocated[array]
      lmu = choose_best(array)
      allocated[array] = lmu
    fi
  od
fi
if ∃ a: allocated[a] == 0
  allocate_rest(allocated)
fi
```

Figure 4.9. Algorithm for Logical Memory Allocation

efficiently.

The most critical array pair are taken from the head of the list. The first task is to select which of the pair is to be allocated before the other. The criteria for selection are:-

- Choose the array with the greatest "net weight", that is the sum of all the array's entries in the allocation matrix. If both weights are equal,
- Choose the array with the greatest "allocated weight", the sum of the entries for the array which correspond to arrays already allocated. If both these

weights are equal then,

- Choose an array at random from the pair.

The aim of the first two tests is to select the most critical array between the two. The first test does this by choosing the array with the most parallel potential with all other arrays in the program. The second looks for the array with the most potential with those already allocated of the two. The most critical array, therefore, is that one which is most likely to benefit the schedule by being in separate memory from other arrays. The third test resolves the case of both arrays being equally critical in both the previous cases.

Once the best array of the pair is selected it remains to place it into a memory block. The first array to be allocated is placed in the first memory block. Subsequent arrays will be placed according to their relationship with those already allocated.

The relationship between two arrays is given by their joint weight in the parallel allocation matrix. By summing these weights for each memory in the allocation we get a measure of the value of placing the current array in each memory according to its potential parallel access profile. The memory with the minimum relationship summation is best for the array.

This memory selection will place an array into an empty memory, if possible, otherwise into a memory that ensures the minimum potential parallelism is wasted. Once all pairs on the priority list are allocated the algorithm checks to see if any arrays remain unallocated, and if so it proceeds with serial logical allocation.

Serial Allocation

The procedure followed here is similar for the parallel case. It uses the serial allocation weight matrix. This is a reflection of the number of occurrences that one array has to await the access to another before the first array's access can proceed. Essentially, then, it is a measure of the algorithm's serial nature, the limit on the inherent parallelism of the user's application.

A priority list is compiled for those arrays with a serial relationship. This time the list items contain a single array and its entire "net serial weight". This

is analogous to the "net parallel weight" described above, in Section 4.1.5. The priority list is sorted in decreasing order of net serial weight.

Each array taken from the top of the priority list has its relationships with allocated arrays calculated, as in the previous, parallel case. The memory which contains the maximum serial weight for the current array is chosen for the allocation. This gives an optimal allocation as accesses to the allocated array will have to succeed those of other arrays in the same memory more of the time than for other memories in the allocation. The limits set by the semantics of the algorithm are therefore used in the allocation of arrays to memory.

4.1.6 Rescheduling

Once logical allocation has been performed for all global schedules, with all combinations of memories tried, the task of adjusting the optimal schedules obtained for each block in Section 4.1.2 is performed. These optimal schedules assume infinite resources available, and that scheduling is only limited by the algorithms' inherent data precedences. As a number of architectures now exist, the resources are now bounded and the optimal schedules can be adjusted accordingly.

The rescheduling uses the optimal schedules obtained in Section 4.1.2 as priority lists for the statements in the basic blocks. The rescheduling algorithm used in CSiC in shown in Figure 4.10. It takes a resource list resource_used which gives the memory unit used by each statement in the schedule. The algorithm returns a list of sets of statements representing the adjusted schedule. The statements of each state can be executed in parallel for the particular memory architecture.

In the rescheduled schedules, time steps contain statements of equal priority. No statement in time step $n+1$ of the optimal schedule can be rescheduled until all statements in time step n have been rescheduled. Remembering that the optimal schedules exploit all possible parallelism, mixing statements from two adjacent optimal time steps in the original schedule would render the rescheduled scheme invalid for the original algorithm. Once the blocks have been rescheduled for all synthesised architectures, each architecture can be timed and a cost value calculated.

```
done = false
ostep = first optimal step
rstep = rescheduled first step
deferred = φ
while !done do
 done = true
 ∀ r ∈ resources do
  resources[r] = notbusy
 od
 if !deferred.empty()
  cstep = deferred
  deferred = φ
 else cstep = ostep
 fi
 ∀ s ∈ cstep do
  r = resource_used[s]
  if (resources[r] != busy)
   resource[r] = busy
   step.insert(s)
  else
   deferred.insert(s)
  fi
 done = false
 od
 if deferred.empty() ostep = next optimal step
 fi
 rstep = next rescheduled step
od
```

Figure 4.10. Algorithm for Rescheduling a Basic Block

4.1.7 Improving the Schedules

The scheduling performed by CSiC at present does not pipeline loops, as described in [61] and [62]. These techniques take advantage of the overlap that often exists between adjacent loop iterations, where operations performed at the end of a loop body can be interleaved in the schedule with operations performed at the start.

Such techniques typically require additional initialisation of variables to be added to the algorithmic description before the loop begins, in order to prime the pipeline. Additional operations might also have to be added after the loop has completed, in order to empty the pipeline. Similar additions have to be made by CSiC for some of the transforms that are applied to the input description[4]. The extension of CSiC to implement loop pipelining would be relatively easy as this functionality could be utilised.

Further development of the CSiC tool would require further research into this area. At present the tool produces "unoptimised" schedules , relying on the improvement of memory access to ameliorate the performance of the user's description. Tightly nested block schedules (where a single block makes up the loop body) provide the simplest case for pipelining. Where there are multiple blocks nested in a loop the analysis becomes more complex.

1	A_1 A_2
2	A_3
3	A_4 A_5
4	A_6

Figure 4.11. Example Schedule for Pipelining

In the first case, a simple extension to CSiC would be necessary. An additional stage could be added to the rescheduling process, whereby the head and tail of tightly nested blocks is checked for overlap potential. The dependences between access statements would provide the necessary information concerning the limits of movement for the accesses in the schedule. In order that the schedule

[4]See Section 5.3

maintained the semantics of the original input description, dependences would have to be maintained.

For example, if the basic block schedule of Figure 4.11 were to be tightly nested, pipelining of the loop would involve checking for access conflicts between accesses A_6 and A_1 and A_2. Conflicts would occur if any of the following dependences existed,

$$A_6 \; \delta \; {}^{f,*,o}_{*,\ldots,-1} \; A_1$$

$$A_6 \; \delta \; {}^{f,*,o}_{*,\ldots,-1} \; A_2$$

The distance vectors of these dependences indicate that the relation is carried over a single loop iteration. The dependence source (in this case A_6) cannot move past the sink (A_1 or A_2) in the schedule, without causing a change in the behaviour of the schedule. If the dependence were to be carried over multiple iterations, indicated by a distance of greater than one, or the relation were to be an input dependence then the source could move past the sink in the schedule. In the case of Figure 4.11 this would mean a shortening of the overall schedule, improving the performance.

The absence of dependences between the bottom and top of the schedule would allow overlap to be effected, giving the improvement in performance of the final schedule. The existence of dependences that still allow loop pipelining to take place, either input dependences, or dependences carried by more than a single iteration, would then depend upon the relationship between the dependence source and the accesses performed in the control step following the dependence sink. Conflicts might still occur here, requiring the analysis to be performed again for the next step.

The above describes a strategy for loop pipelining using dependences for finding conflicts. Resource conflicts could also occur, preventing the compaction of an iteration schedule, and these would have to be tested for before a scheduled operation could be moved "floated" from the bottom to the top.

For loosely nested blocks (having more than one block in a loop body), further analysis would have to be performed. The Perfect Loop Pipelining technique, described in [7], will pipeline a loop containing conditionals by successively unwinding the loop and compacting the operations within the extended

body, using inter-operation dependences to limit the amount of compaction. A repetitive pattern is eventually found in the unwound body, usually after $k + 1$ loop-unwinds, were k is the length of the longest path through the body of the loop [7].

This technique is utilised in Percolation scheduling [61], covered in Section 2.1.3 for generating optimal schedules (in control steps) which are then bound to the resources available. The techniques described earlier in this section show how CSiC uses a similar method for scheduling memory accesses for the synthesised architectures.

4.2 Timing the Architectures

Once synthesis has been completed for all the global schedules a timing estimate for each is calculated. This measure is used to give some measure of the likely performance of the solutions. Once timing and costing[5] is complete, the user can make a selection between the different synthesised architectures, or choose to try and improve the solutions created by applying high level transforms to the algorithm. These transforms are covered in Chapter 5.

```
∀ global schedules, S_i^G do
  ∀ synthesized architectures, A_j^i do
    using local schedules, S_{i,j}^L
      T_{i,j} = time_arch(S_i^G, S_{i,j}^L)
  od
od
```

Figure 4.12. Traversal of Data Structures for Timing

The results of the scheduling and allocation phases, detailed in the previous sections, are a number of different architectures for each global schedule, with a set of local schedules for the basic blocks of the algorithm. These structures are used as shown in the algorithm of Figure 4.12.

[5]See Section 4.3

```
procedure time_arch()
 // passed a super-node in a global schedule, S_n^G,
 // and a set of local schedules, S^L,
 // returns a worst case integer timing value, t
 t = 0
 t_c = 0
 // L gives ordering of sets, sets contain parallel nodes
 L = list of sets of super-nodes in S_n^G
 ∀ sets, s_i^P ∈ L do
  t_s = 0
  ∀ super-nodes, v ∈ s_i^P do
   if is_block(v)
    t_v = length of block schedule in S^L
   else
    // recursive call on v
    t_v = time_arch(S_v^G)
   fi
   if t_v > t_s
    t_s = t_v
   fi
  od
  t = t + t_s
 od
 // check for super node, n, being if-then-else
 if is_conditional(S_n^G)
  L_c = else-part of S_n^G
  repeat above for L_c, leaving result in t_c
  if t_c > t
   t = t_c
  fi
 fi
 // I_n is iteration count for super-node n
 total = I_c × t
 return total
end
```

Figure 4.13. *time_arch* procedure details

All architectural combinations are timed in an iterative manner and their results stored in a timing matrix. The timing calculation executes in times about equal to the parse time for the input description. Obviously there is some variance between different inputs, as each will depend upon the number of global and local schedules, and the number of arrays in the user algorithm

The actual timing procedure, time_arch(), returns a single figure which is a worst case timing for the architecture. The basic algorithm for this procedure is shown in Figure 4.13.

The time_arch() procedure performs a recursive depth-first traversal of the global schedule. Recalling from Section 4.1.1 that the global process schedules are lists of sets of super-nodes, which are executable in parallel in the order given by the list, and that each super-node contains another list of sets, or else represents a block node. The block super-nodes provide an index into the local block schedules for the specified architecture. The procedure reaches these blocks which provide the basic timing information, given by the length of the block schedule. Each super-node has an iteration count, which is taken from the loop information extracted earlier on[6].

After the contents of each super-node has been timed for a single iteration, its total execution time is obtained by multiplication of the worst case timing by the iteration count. The worst case time is obtained by taking the maximum execution time for each super-node in each parallel set. In the case of the super-node representing a conditional if-then-else structure, the longer time for the two branches is taken.

Selection of the worst case timing is a conservative measure. No control flow analysis for establishing relative execution of conditional branches is performed by CSiC . Further work would be required to allow the user to annotate conditional branch statistics onto the input description. This would allow for more accurate timing calculations to be performed, which would give a better characterisation of the hardware performance, although not necessarily improving the selection between different architectures.

[6]See Section 3.4.5

4.3 Costing the Architectures

In order to cost the various solutions that CSiC generates there need to be relative costs for the different types of memory allocated. The basic memory building blocks are RAM and shift-registers. RAM can be both on and off chip. The following cost plan is for silicon real estate on a custom chip designed by the compiler.

4.3.1 On Chip RAM Cost

For a RAM block with W words of B bits, the cost, $C_R(W, R)$, is defined as

$$C_R(W, B) = \alpha_R WB + \beta_R(W) + \gamma_R B$$

The first term refers to the cost of the RAM cells. The constant, α_R, is related to the number of transistors in a RAM cell. The second term in the equation is an expression of the addressing cost of the RAM block.

The function $\beta_R()$ assumes that address sequences are fairly regular (commensurate with arrays accessed from nested loops) and derives from work detailed in [107]. This shows that a semi-random sequence of bits, which repeat every 2^N addresses, can be generated by n counter bits plus minimal logic. Thus the function $\beta_R()$ is derived as

$$\beta_R(W) = C_S(log_2 W, 1)$$

The cost function, $C_S(W, B)$ is given in the next section.

The final constant γ_R expresses the cost of the sense amplifiers at the column ends of the RAM array. This is again an expression of the number of transistors contained in the amplifier.

4.3.2 On Chip Shift Register Cost

For a shift register with W words of B bits, the cost $C_S(W, B)$ is taken as,

$$C_S(W, B) = \alpha_S WB + \beta_S W + \gamma_S B$$

Where α_S is the cost of a flip-flop cell, β_S is the cost of inter-cell connections and γ_S is the cost associated with stacking flip-flop cells on top of one another. This final coefficient is minimal and is taken to be 0.

At present CSiC does not have support for different types of memory on or off chip. Logical Memory Blocks are allocated and assigned physical quantities of memory at a later stage. The cost of the memory allocated is taken as the size of the memory block. The addressing and other costs associated with memory are not accounted for by the tool.

4.4 Summary

Methods have been presented for utilising the information obtained from the input description and described in Chapter 3. These methods provide a basis for the memory synthesis task; schedules for all access statements within the program are generated and the data which they load and store is assigned storage. This arrangement of data and statement is optimal for the resources available, in terms of memory access.

An interleaved scheduling and allocation method was presented which attempts to distribute data amongst a number of different resources so as to obtain maximum parallelism in the access of that data. The allocation phase utilises data from both the local (basic block) and global (loop process) levels. This data is obtained from the respective optimal schedules. Once allocation is performed the schedules are re-adjusted for the resources provided.

Methods for the timing and costing of the architectures produced were also presented. These provide the CSiC tool with a means of evaluating the solutions produced.

Chapter 5

Transforming the Code

This chapter covers the high level transforms implemented by CSiC in order to improve the internal description held for synthesis. The transforms operate directly on the Statement Flow Graph (SFG[1]); this ensures their correctness can be properly validated, as CSiC can emit the SFG's C code description before and after the transform is applied.

The presentation here is of array based transforms that improve the memory architecture represented by the C code description given to the CSiC tool. This is achieved by attempting to reduce the access bottleneck through the reordering of statements in the SFG and by their transformation. The transforms are dependence based, that is they rely on array dependence information for their application. Much of the early work on dependence analysis was performed by the super-optimising compiler community [75, 77, 76]. The use of this dependence information for the transformation of source code has been the subject of much research [108], [84], [109], [110], primarily for the optimisation of Fortran program loops, for running on vector processors.

The transforms presented here concentrate on reducing the array access within loops, rather than the restructuring of loops. This latter task is beyond the current scope of this research, techniques already reviewed indicate the validity of this type of transform in general synthesis. It's applicability to memory synthesis is undoubtably as valuable.

[1] see Section 3.4.1

5.1 Transforms In CSiC

The transforms are presented in approximate order of application. During each synthesis iteration, CSiC collects all those transforms that might be applied to the SFG, along with the access saving potential of each and an estimation of the associated cost. The transforms are then applied to the CFG in an iterative manner, optimising the source description by reducing the access count represented by array accesses.

Array based transforms operate on array statements in the code description. The transforms use the dependence relationships found between different statements in the input at earlier stages of the analysis[2]. Dependence between two array statements implies that they access the same memory location at some time during execution. This represents an access overhead that might be unnecessary, examination of the information associated with the dependence can be used to ascertain this. In the case of an unnecessary overhead in accessing, the source code can be transformed in order to remove this redundancy, whilst still maintaining the original semantic meaning of the code.

These are transforms that can be applied directly to the statement flow graph. Certain conditions have to be met before a particular transform can be applied. The original sense of the algorithm is preserved by the transforms. Because they operate directly on the statement flow graph they can be thought of as high level transforms because they provide an improvement on the architecture originally "specified" by the designer's algorithmic description.

The transforms introduced in this Chapter are named as follows,

- SRRT - Simple Redundancy Removal Transform
- CRRT - Carried redundancy Removal Transform
- AMT - Access Motion Transform

The first transform introduced can be thought of as a special case of the second. It is simpler to implement and produces a minimal impact on the total

[2]See Section 3.5.1

memory cost, introducing only extra scalar variables to the description. However its efficacy can be considerable and it plays an important part in transforming the source. The second transform can have a potentially greater impact on the algorithmic performance as it introduces intermediate memories into the algorithm which perform a cache-like function. The third transform is similar to the first in terms of its impact on the memory cost, however its nature is closer to the second.

5.1.1 On Notation

The dependence analysis techniques reviewed in Section 2.3.1 were accompanied by an introduction to the notation used in the literature. In the following sections I keep with this notation. Dependences are represented as

$$S \; \delta \, {}^{t}_{\mathbf{d}} \; T$$

where S and T are the array statements in question. They are related by one of the following,

S	$<$	T	S comes before T in the input description
S	$>$	T	S comes after T
S	$==$	T	S is the same statement as T

The dependence type, t, can be flow (t = f), input (t = i), output (t = o) or anti ($t = *$). The direction or distance vector, $\mathbf{d}$, will usually use a distance value, except in cases where the distance is unknown or irrelevant.

$$\mathbf{d} = (d_1, d_2, \ldots, d_k)$$

where k is the number of levels, that is loops, that nest the statements, and

$$d_j = \begin{cases} -n & \text{or} \quad < \quad i_S - i_T = -n & \text{or} \quad i_S < i_T \\ 0 & \text{or} \quad = \quad i_S - i_T = 0 & \text{or} \quad i_S == i_T \\ +n & \text{or} \quad > \quad i_S - i_T = n & \text{or} \quad i_S > i_T \end{cases}$$

where i_S is the induction variable's value for statement S and i_T is the induction variable's value for statement T, at level j.

5.1.2 Dependence Types

Of the four types of dependence (flow, input, output and anti) that can exist between array statements, only three are directly relevant here. These are flow, input and output dependences. Anti-dependences do not indicate a transform can take place, in fact they inhibit transformation, as will be seen later, see Section 5.5.

The three types of interest here all indicate some kind of redundancy between two statements: an input dependence shows the same location is read by both statements; an output dependence indicates the same location is written by both; a flow dependence indicates that the second statement reads the location written by the first. An anti-dependence indicates that a location is successively read and then written and are used here to break a sequence of transforms.

The transforms presented here aim to remove the redundant accesses between statements and across different loop iterations. As will be shown later, this leads to more optimal synthesis results.

5.1.3 Redundancy Removal

Redundancy removal involves the identification of instances in the code where multiple accesses are made to the same memory location, resulting in an unnecessary overhead in communications throughput. In CSiC this situation is detected by analysing the dependence relationships between statements that access array data. Transforms are then applied to the internal source description of the user algorithm, the SFG. These remove the unnecessary redundancy in the memory I/O and lead to a more optimal description from which to synthesis hardware.

Two flavours of redundancy removal are described below: Simple and Carried. The difference between the two comes from the type of dependence relation between the statements being transformed. Where there is a loop independent dependence, Simple Redundancy Removal (SRR) is used; for loop carried dependences, Carried Redundancy Removal (CRR) is performed.

5.1.4 Access Motion

The other type of transform used by CSiC is called Access Motion. This is akin to loop-code-motion, a well known code optimisation technique. The difference is that where loop-code-motion removes repeated calculations from within a loop body, Access Motion removes repeated, or redundant, accesses from the loop body. This serves to optimise the memory communication by reducing the total access count.

Access Motion is similar to Redundancy Removal, and acts in much the same way. The type of access dependence that identifies a potential access motion transform is slightly different, and so a distinction is made. All three transforms are described in the following sections.

5.2 Simple Redundancy Removal

To illustrate the action of the SRRT and to show its relation to statement dependences a number of code fragment examples are examined below. The examples come from the test cases considered by the author in Chapter 6.

Example 0: Simple Case

A common occurrence in user code is the situation where a dependence occurs between two statements enclosed by the same loop, for example

```
for (i = 0;i < I;i++) {
  for (j = 0;j < J;j++) {
    A[i][j] =  ... ; /* S */
           ...
     ... = A[i][j];  /* T */
  }
}
```

The dependence here is $S\ \delta\ ^{i}_{=,=}\ T$, and $S < T$. It is assumed that there is no other statement between S and T that is dependent upon S. The code shows that statement T reads the same location as statement S within the same loop

iteration. This represents an access overhead when the code is synthesised by CSiC , resulting in a schedule length that is longer than it need be. The SRRT replaces the second load from the array with a local variable that is set to the contents of the memory location. The transformed C code now looks like,

```
for (i = 0;i < I;i++) {
  for (j = 0;j < J;j++) {
    _tmp =  ... ;
    A[i][j] = _tmp;  /* S */
        ...
     ... = _tmp;     /* T */
  }
}
```

The above example is for a flow dependence. In the case of input and output dependences the example would be very similar. For an input dependence between S and T a local variable would be set to the value of the read in S. The read access in T would then be replaced by this local.

In the case of an output dependence between S and T, of the same distance as before, the write access of S would simply be removed from the code, leaving statement T to write the array location. Note that the same assumptions apply, namely that there is no statement between S and T that is dependent upon S.

Example 1: Across A Loop Boundary

This example illustrates the case where a dependence is carried across a loop boundary, but is still only carried across a fraction of a loop iteration. This is the result of the relative position of the two statements, with $S > T$.

```
for (i = 0;i < I;i++) {
  for (j = 0;j < J;j++) {
    for (k = 1;k < K;k++) {
      ... = A[j][k-1]; /* T */
                ...
      A[j][k] = ...;   /* S */
  }
}
```

Here the dependence is $S\ \delta\ ^{I}_{0,0,-1}\ T$. At the start of each iteration , T reads a value written in the previous iteration by statement S, as indicated by the dependence distance of -1 between the two statements. In order to transform the code here, it is necessary to replace the access in statement T by the local variable set to the contents of the memory location addressed by S. An extra access has to be introduced in order to initialise the local before the inner loop is executed. The transformed source is then,

```
for (i = 0;i < I;i++) {
  for (j = 0;j < J;j++) {
    _tmp = A[j][0];
    for (k = 1;k < K;k++) {
      ... = _tmp;    /* T */
              ...
      _tmp = ...;
      A[j][k] = _tmp;       /* S */
  }
}
```

Example 2: Across Basic Blocks

This example illustrates the case of S and T residing in different blocks. This situation occurs for conditional statements embedded into loops and also for statements present in different, but shared, levels of loop nesting. The following code fragment provides an illustration,

```
... = a[i];     /* S */
      ...
if (condition)
{
      ...
   ... = a[i];  /* T */
}
 ...
```

Here the dependence is $S \; \delta \stackrel{i}{=} \; T$ and $S < T$. However T is in a different block in the SFG to that of S and so any substitution of the value read by T must be assured to be the same as that for S.

In this case the transform can take place as for Example 0 as any execution of statement T will *always* be preceded by execution of statement S, therefore the dependence is guaranteed to hold.

A different potential ordering for S and T is,

```
if (condition)
{
      ...
   ... = a[i];   /* S */
}
    ...
 ...  = a[i];    /* T */
    ...
```

This time there is no guarantee that execution of T is necessarily preceded by execution of S. For this case the transform could not proceed as before, substituting a temporary variable for the read in T, set by the read in S. Instead the accesses are left as they are and no optimisation can be performed without further prior transformation of the source code.

From this simple example, easily generalised for the others presented above, the transform can only be applied if the source statements execution can be guaranteed when altering the sink statement for the dependence. The concept of *dominance* was introduced in Section 3.4.5, and its implementation into the CSiC system described. This graph technique provides a method of checking that the above criteria are met, before applying the transform. Guaranteed execution of S preceding execution of T only occurs when statement S *dominates* T in the SFG (or S's basic block dominates that of T).

In some cases it will be possible to move the read in S and T in the last example so that it appears before the conditional block. This will guarantee the removal of the access within the conditional body. In order to be able to do this there must exist no dependences between either S or T with any other statements between their original positions and the position of the new access in the transformed code.

5.2.1 Defining the Transform

The examples presented above describe the application of the SRR Transform in a number of different situations. These describe the transform's behaviour sufficiently enough to define it. To summarise, the Simple Redundancy Removal Transform (SRRT) uses dependences between statements which are carried for a fraction of a loop iteration. The source statement must dominate the dependence sink statement for a valid application of the transform to be achieved. A fraction of a loop iteration implies that control moves from the source to the sink of a dependence without passing the source again. This occurs in two instances,

1. The source statement, S, comes before the sink, T, in the syntactic order of the loop body, $S < T$, and the dependence is not carried by the loop, $\delta_{0,\ldots,0}$.

2. The source, S, succeeds the sink, T, in the syntactic order of the loop code, $S > T$, and the dependence is carried a unitary distance by the loop, $\delta_{0,\ldots,-1}$.

Thus the transform can be defined by:-

Transform 1 (SRRT) *Given two statements S and T whose only dependence relations are either*

$$S\ \delta^{[iof]}_{0,0,..,0}\ T \text{ for } S < T \text{ and } S \text{ dominates } T$$

or

$$S\ \delta^{[iof]}_{0,0,..,-1}\ T \text{ for } S > T \text{ and } T \text{ dominates } S$$

then, in the case of a flow or input dependence the dependence's sink load can be replaced by the scalar reference set in the dependence source; in the case of an output dependence, the dependence's source store can be removed from the code. In the case of the second condition above being true, the scalar should be appropriately initialised in the case of an flow or input dependence.

Once dominance has been established for the source over the sink the two cases of intra-block and inter-block transformation are treated in the same manner.

It is important that no other dependences that alter the same location, exist between S and other statements before T is reached. Any that are present do not affect the transform if they are of the same type as that between S and T. The strategy in CSiC is to examine statements with no such dependences between them first.

In the event of statement T being at a higher level than S a dependence will not be picked up (see Section 3.5.1) as dependence is searched for only in the case of statements being at the same level. The Access Motion Transform, see Section 5.4, can be used to move a redundant access statement from a higher to a lower level. This will expose opportunities for SRRT to be applied at a later stage, with statements moved to the same level.

The SRRT can be used for simplifying the analysis and synthesis of conditional constructs. If the same access statement occurs in both branches of a loop-nested if-then-else statement then it can be moved out of the construct.

5.2.2 Costing the Effect of the Transform

The two cases where the transform can be applied each have a different effect upon the access overhead of the transformed code. In the first case, where

$$S \; \delta^{[iof]}_{0,0,..,0} \; T \text{ for } S < T \text{ and } S \text{ dominates } T$$

a straight saving is made which is equal to the number of iterations of the loops that enclose the statement that is altered. Thus for a SRRT dependence sink at level i, the saving in accesses is,

$$\prod_{k=1}^{k=i} I_k$$

where I_k is the number of iterations at level k.

For the second case, where

$$S \; \delta^{[iof]}_{0,0,..,-1} \; T \text{ for } S > T \text{ and } T \text{ dominates } S$$

a lower access saving is obtained than for the first case. The saving of the transform now becomes,

$$\prod_{k=1}^{k=i-1} I_k(I_i - 1)$$

comparable to the previous case for $k >> 1$.

5.3 Carried Redundancy Removal

The previous section introduced the SRR Transform which makes use of dependences carried across a fraction of a loop iteration. This reduces access redundancy by storing memory values in scalar variables, for storage in registers, for a fraction of an iteration. The Carried Redundancy Removal Transform (CRRT) utilises dependences that are carried across one or more loop iterations, requiring a different memory scheme for storage of the intermediate values.

The SRRT increases the local variable count, and it is assumed that this overhead is dealt with at the data path synthesis stage. It is anticipated that the output from CSiC would be directed to such a synthesis tool which would

perform register allocation and optimisation, as in [11] and [111]. Whilst the transform increases the number of local variables, which will produce some additional register pressure, there will not necessarily be a linear rise in the number of registers required. This is due to the lifetimes of different variables within the code description, many of which will not be overlapping, representing a free register resource during these times.

The CRRT introduces an intermediate level of storage that resides between the main memory and the processing core. Typically this extra level comprises shift registers, which implement data delays within the loop lifetime. However, the allocation of shift registers will be dependent upon the distance the dependence is carried across the loop. For dependences with small distances, the intermediate storage might be in registers on the data path, represented by scalar variables in CSiC .

The current implementation of CSiC represents these delays as shifts of local variables, in a scheme similar to that found in [85]. The shift itself is represented as a single scalar variable, all that is required for the timing analysis currently performed by CSiC . Further development of the tool would involve the definition of different types of memory storage that would be handled by the scheduling and allocation phases. This would require a simple check on the declaration of an array variable before allocation[3]. This check would indicate the type of storage required for the variable, and what other arrays could share the storage.

Some examples for the CRRT are presented below and then the transform is defined.

Example 3: Simple Case

A carried dependence in the user code indicates that a memory location is accessed during one loop iteration and re-accessed in a later iteration. This occurs in the code fragment

[3]See Section 4.1.5

```
for (i = 2;i < I;i++) {
  A[i] =  ... ;      /* S */
     ...
    ...  = A[i - 2]; /* T */
}
```

The dependence in this example is

$$S \; \delta \; {}^{I}_{-2} \; T$$

which indicates that T reads a value from memory written by S two iterations previously. In order to remove this redundancy it is necessary to introduce a shift function into the code which carries the values across the loop iterations. The number of local variables required is $n + 1$, where n is the dependence distance. The shift has to be initialised before the loop is executed so that the first n iterations read the correct values. The transformed code is shown below.

```
_tmp0 = A[0];
_tmp1 = A[1];
for (i = 1;i < I;i++) {
  _tmp2 = ... ;
  A[i] = _tmp2;  /* S */
      ...
    ... = _tmp0;  /* T */
  _tmp0 = _tmp1;
  _tmp1 = _tmp2;
}
```

A method for performing this type of transform is given in [85], used for software compiler optimisation. When the dependence type is flow or input, the method is the same; replace the sink of the dependence with the appropriate local variable. In the case of an output dependence, the situation is slightly different.

If there exists a flow or input dependence between the source, S, and another statement, say U, with $S < U < T$, then the value written by S should be loaded into the shift chain. This value can then be assigned to U at the appropriate point. Finally T is left to perform the write.

The above example illustrates the use of multiple scalar variables to represent the shift. As mentioned above, CSiC does not currently implement shift registers in its memory types. The transform can equally be represented as a single scalar variable, its effect on the timing analysis performed is the same. Further development of the tool would require the representation of shift registers as special arrays that would be allocated differently from the currently recognised types.

Example 4: Multi-dimensional Arrays

I now examine the case of having greater dimensioned array statements that can undergo the transform, and the implications that this has for the memory architecture. The code fragment below illustrates a typical example found in code samples. An array access is located within deeply nested loops, its index variables are incremented in loops outside the scope of the most tightly nesting loop. This represents a significant access overhead for an algorithm as the same values are accessed repeatedly during the loop lifetime.

```
for (i = 0;i < I;i++) {
  for (j = 0; < J;++) {
    for (k = 0;k < K;k++) {
      for (l = 0;l < L;l++) {
        for (m = 0;m < M;m++) {
          for (n = 0;n < N;n++) {
            ...
            t = A[i][m][n]; /* S */
            ...
          }}}}}
```

The following dependences apply to statement the S, referencing a three dimensional array A. The statement is nested by six loops. Not all the dependences for the statement are shown, only those carried by the loops.

$$S\ \delta^{i}_{=,=,<,<,=,=}\ S \quad S\ \delta^{i}_{=,=,<,=,=,=}\ S \quad S\ \delta^{i}_{=,=,=,<,=,=}\ S$$
$$S\ \delta^{i}_{=,<,<,<,=,=}\ S \quad S\ \delta^{i}_{=,<,<,=,=,=}\ S \quad S\ \delta^{i}_{=,<,=,<,=,=}\ S$$
$$S\ \delta^{i}_{=,<,=,=,=,=}\ S$$

The statement is at level 6 and is referenced using the induction variables from levels 1,5 and 6 (i, m and n respectively). We see that for each iteration of the outermost loop there are $J \times K \times L$ accesses to each datum held in the portion of A represented by $A[i]$. This represents a redundancy of

$$\prod_{k=2}^{k=4} I_k - 1$$

for each element in the array portion represented by $A[i]$. This leads to a total redundancy of

$$\prod_{k=5}^{k=6} I_k (\prod_{k=2}^{k=4} I_k - 1)$$

for each iteration of the outer loop.

In the example given above the CRRT dependence is $S\ \delta^{i}_{=,<,<,<,=,=}\ S$. The various regions in the dependence direction are indicated in Table 5.1.

=	< < <	= =
outer level(s)	carrying level(s)	redundant level(s)

Table 5.1. Regions of the SRRT Dependence

The outer levels indicate the loops that remain in the same iteration throughout the dependence. This is indicated by the '=' sign in the direction vector. The carrying levels show the loops that carry the dependence. The dependence exists between each iteration of these loops, creating the redundant accesses. The redundant levels show those loops that cause the dependence on the array location.

Each iteration is dependent upon the previous iteration of any of the carrying loops.

In order to remove this redundancy the code has to be transformed so that the inner loops at levels 5 and 6 make an access to each data item in $A[i]$ just once and that the data is held locally from then on. The local storage required is dictated by the iteration count of the loops whose induction variables are used to access the array to the right of the '<' sequence in the direction vector. In the example above these are the loops at levels 5 and 6. The amount of storage in this case is $M \times N$.

The local storage takes the form of a circular shift register, assumed to be incorporated into the processing core in the final integrated circuit. This intermediate memory structure provides fast access to the array data which does not require an external memory operation, thus decreasing the processing time for the code section.

CSiC does not currently implement a mechanism for the representation of shift registers or methods for specifying where in a design specific memories are located, the addition of these shift registers is affected by regarding them as local scalar variables in the code. The effect of this is that their cost cannot be taken directly from the transformed description, but the tool will nevertheless be able to calculate their effect on the overall performance of the transformed algorithm. This is because the array accesses are removed, simulating the effect on the hardware of the addition of the shift registers.

The code fragment below shows the transformation performed for the example given above. A separate process has been inserted at level 1 which fills the shift register. This shift register is then read in place of the original access, giving no access overhead for the inner loop, thereby increasing the throughput of the fragment.

Example 5: Use of Guards

This representation will not produce the most efficient implementation, as operations performed in the body of the level 6 loop could proceed in parallel with the filling of the shift register. An alternative transformation would involve the inclusion of guard statements in the inner loop which ensured the first $M \times N$

```
for (i = 0;i < I;i++) {
  for (t1 = 0;t1 < M;t1++) {
    for (t2 = 0;t2 < N;t2++) {
      shiftMN = A[i][t1][t2];}}
  for (j = 0; < J;++) {
    for (k = 0;k < K;k++) {
      for (l = 0;l < L;l++) {
        for (m = 0;m < M;m++) {
          for (n = 0;n < N;n++) {
            ...
            t = shiftMN; /* S */
            ...
          }}}}}}
```

iterations filled the shift registers and provided data for the loop body's execution. The guards would ensure that further iterations access the shift register variable, rather than main memory.

Thus, the first part of the transformation would be the insertion of control flow. Assuming the variables k_o, k_c, k_d indicate the levels at which start the outer, carrying and redundant levels respectively, and that k_S is the level of the access statement, the method used would be

1. $\forall$ statement S, with CRR dependences
2. insert guard activation before loop at level $k_c - 1$
3. replace S at level k_S with guard check
4. insert guard deactivation after loop at level $k_r - 1$
5. insert appropriate code into guard check options

The guard assignment indicates that a new iteration of the outer loop has begun and that new data must be loaded into local storage. This is represented in the internal description as a variable assignment. This variable is tested at each iteration of the (most inner) redundant level, level k_S. If the guard is true then

data is loaded from main storage, otherwise the local storage is interrogated for the data. At the last level of the carrying region, $k_r - 1$, after the redundant loops in the code the guard deactivation statement is inserted. This indicates that all data has been loaded into local storage and further iterations of the redundant levels can obtain data from here. This would result in a transformation looking like,

```
for (i = 0;i < I;i++) {
  guard = 1;
  for (j = 0; < J;++) {
    for (k = 0;k < K;k++) {
      for (l = 0;l < L;l++) {
        for (m = 0;m < M;m++) {
          for (n = 0;n < N;n++) {
            ...
            if (guard) {
              shiftMN = t = A[i][m][n];
            }
            else t = shiftMN;
            ...
          }
        }
        guard = 0;}}}}
```

This guard scheme is not used at present as the timing algorithm in CSiC [4] conservatively assumes that the most expensive conditional code, in terms of memory access, is always executed. Thus to the timing algorithm the transformed code would have the same performance as the un-transformed source. Future development of CSiC would take this into account and ways of communicating the number of executions of such known conditional bodies would be developed.

[4] See Section 4.2

5.3.1 Defining the Transform

The conditions on the application of the CRRT are similar to those for the SRRT. Anti-dependences do not represent an access overhead, we are only interested in flow, input and output dependences. The CRRT characterisation is,

Transform 2 *CRRT The transform is applied when dependences of the form*

$$S \; \delta \; {}^{i,o,f}_{0,0,\ldots,-n} \; T \text{ for } S < T$$

are found. The distance, n, of the dependence and the dimensionality of the array involved determine the number of local variables that are required for the transform.

Similar conditions apply to the determination of dominance of the dependence sink by the source as for the SRRT. Similarly, other dependences that involve either S or T have to be taken into account.

CSiC currently detects such instances where the CRRT is applicable, but does not actually perform the transform itself. This has to be done by hand and the resultant input description fed back into the tool. Implementing the transform would not represent a large task however. An algorithm for performing the SRRT and CRRT is outlined in Section 5.5. This is currently in place for the SRRT, and could be easily extended to perform CRRT.

5.4 Access Motion

Access motion is a form of loop code motion [52] that uses access dependence information to remove unnecessary array references from inner loops. This works in much the same way as loop code motion which removes general expressions from within a loop, where the result of the expression does not change from iteration to iteration. In the case of loop code motion expressions whose results do not change during the lifetime of the loop are evaluated before the loop body, thus reducing the time to execute the loop.

The Access Motion Transform (AMT) is similar in that it removes accesses

from within loops which access the same memory location each time; thus representing a redundant access in each iteration. Moving the access to a position outside the loop body and replacing each reference to the access with a local variable containing the location's contents will remove this redundancy and improve the I/O throughput.

Detecting situations where the AMT is applicable involves searching for statements that have dependences on themselves. As statements in CSiC 's internal representation are in triplet form (see Section 3.4.2) we are only concerned with input or output self-dependences. There is a slight difference in the handling of the two, this will be explained at the end of the section. An example serves to illustrate the action of the transform.

Example 6: Single Dimensioned Array

The array A in the example fragment below is single dimensioned for simplicity, although the transform is also applicable to multi-dimensional arrays.

```
for (i = 0;i < L;i++)
{
  /* level 1 */
  for (j = 0;j < N;j++)
  {
    for (k = 0;k < M;k++)
    {
        ...
      tmp1 = A[i]; /* S */
        ...
    }
  }
}
```

The statement S is dependent upon itself, each repeat iteration of the two inner loops results in a read of the same location. This is the redundant accessing that occurs. The AMT aims to remove this redundancy by promoting the access statement to a higher nest level. In this case the statement can be promoted to level 1, assuming no other intervening statement modifies the location, or the

temporary value.

This is a similar method to that used in percolation scheduling [61], except that the latter will not promote operations outside of basic blocks. A number of dependence relations are found from the dependence. These are

$$S\ \delta\ ^{i}_{=,=,<}\ S \quad S\ \delta\ ^{i}_{=,<,=}\ S \quad S\ \delta\ ^{i}_{=,<,<}\ S$$

Of these the third dependence $S\ \delta\ ^{i}_{=,<,<}\ S$ contains the longest stretch of '<' directions, including the most significant level (level 3 in this case). The position of the left-most '<' indicates that the statement can be moved to level 1, inside the first loop. The transformed fragment is given below.

```
for (i = 0;i < L;i++)
{
  tmp1 = A[i]; /* S */
  for (j = 0;j < N;j++)
  {
    for (k = 0;k < M;k++)
    {
        ...
      ... tmp1 \%\ldots\%
        ...
    }
  }
}
```

The effect of this transform is to reduce the number of accesses by

$$\prod_{k=1}^{k=j} I_k(\prod_{k=i-j}^{k=i} I_k - 1)$$

where j is the level of the leftmost '<' in the chosen dependence and i is the level from which the statement is moved, the maximum level in the dependence. So for this example, the transform represents a saving of $L \times (N \times M - 1)$ accesses.

The above example is for an input dependence. As mentioned previously the transform is also applicable for output dependences. In this case the statement would be moved to just after the nest loop j, rather than just before the loop as for the input relation.

In both cases the statement must be checked for dependences with other statements in the loop body. Dependences of the same type as the AM relation (eg input or output) will present no problems, and are detectable by the RR transforms. However, any dependences that prevent an AM type transform (eg an anti dependence for an input AMT), that exist within the same scope as the AMT dependence, will result in the transform corrupting the semantic meaning of the code. Such dependences have to be checked for before the AMT can be confirmed.

5.4.1 Defining The Transform

The transform represents a time-area trade-off whereby the access overhead is reduced but the register count is increased inside the processing core. However, for single variables, register minimisation techniques will make optimal use of registers present in the data path. Hence the effect of the Access Motion transform is not necessarily the introduction of a new register to the final data path, more an increase on the register load during this section of the algorithm.

Transform 3 (Access Motion) *Given a statement S, nested by n loops, and which accesses array A, find the set of loop carried dependences for S, C_S on itself. From this set find the dependence with the longest sequence of '$<$' symbols in the direction vector, including the last level. The leftmost '$<$' in this sequence gives the level below which the statement can be moved to.*

5.5 An Algorithm for Transform Application

In order to make use of the optimisation opportunities detected by CSiC a strategy is required to perform the transformations on multiple statements. This section describes an algorithm for doing so, which is currently implemented for the SRRT

in the tool. It makes use of the definitions described for the transform in Section 5.2.

The examples presented above assume two statements S and T are interdependent in isolation, that is no other statements are present that also have some dependence relation to either. This is a reasonable assumption for some cases, but by no means all. Typically array statements nested within a loop will access the same arrays a number of times, possibly in deeper nests, or else within conditional blocks also nested by the outer loops.

Such statements form clusters of interconnected statements in the SFG, with dependence arcs defining the clusters. The algorithm must be able to apply the transforms to the statements within each cluster in order to optimise the code description. In order to do this the statements must be ordered in some manner so that the transforms can be applied sensibly. The algorithm in Figure 5.1 provides a method for accomplishing this.

```
1Extract cluster graph of SRR dependent stmts from SFG
2Find the source node for the cluster
3Order remaining nodes according to statement order
4Perform SRRT between source and ordered sinks
  Maintain a current local that carries the datum
  If the current sink is a store
    make its node the source
  goto 4
```

Figure 5.1. Algorithm for Applying Multiple SRR Transforms

The algorithm removes the need to sort through the dependences that will exist between all statements in the cluster. The clusters are formed by statements that all access the same array location during the loop iteration. They will consequently all be linked together by dependence arcs.

A cluster's source node is the source statement at the start of the dependence chain. This node will have an in-degree of zero, although there may be anti-dependence arcs incident upon the node. These are discounted in the search for the source. The source will typically be the first statement in the syntactic

ordering, or possibly the last if the fractional loop iteration that carries the dependence begins at the end of the loop body.

During the construction of the clusters tests are applied for dominance between adjacent nodes. A dependence arc can only lead to inclusion into the cluster if the (local) source dominates its sink. This excludes statements from being transformed that would violate the semantics of the program code.

When the algorithm reaches step 4 it will have a cluster rooted by a source node which will be connected to a succession of other nodes, ordered syntactically. Applying the SRRT to the first node in the ordering will result in a reduction of a single access. If the dependence type is either flow or input the algorithm continues to the next greater node in the ordering. In the case of an output or anti dependence the current source is adjusted as necessary and the current sink made the new source. The process then continues.

This changeover removes the need to prune the cluster graph and order the dependences within it. As the statements all refer to the same location there will be arcs between each one. This could produce difficulties in deciding which dependences to use in the transform. The algorithm gets around this by utilising the statement ordering. Breaks occur naturally by transferring the currently carried value from the previous source to the new value written by the code, at the new source's location.

5.6 Conditions on Transformation

The preceding descriptions of the transforms used by CSiC make certain assumptions. The primary assumption being that if the source statement, S is executed, then the sink statement, T will always be executed. This is not necessarily the case in all instances. Consider the following code fragment

It is clear that if statement T executes, then it is safe to assume that statement S will have executed before it. The same assumption can be made in the case of statement U. However it cannot be assumed that T is executed every time control reaches statement U.

When detecting opportunities for the application of the above transforms it is important to check for such situations. Section 3.4.5 introduced the subject

```
...;    /* S */
if (x) {
  ...;  /* T */
}
...;    /* U */
```

of block ordering and dominance. A data flow algorithm was used to obtain the *dominance* relationships between different blocks. Some block u is said to dominate another v, written u dom(v), if control is guaranteed to have passed through u every time it reaches v. In the code example above, S dom(T) and S dom(U), but T does not dominate U.

With respect to the application of the transforms such situations have to be detected. A check is performed upon each pair of dependent statements to ensure that the source dominates the sink, before transformation is allowed. The dominance sets generated by CSiC for each block allow this check to be performed quickly.

5.7 Summary

This chapter has introduced high level transforms utilised by CSiC for improving the memory architectures generated by the tool. These transforms are high level as they operate on the source level description provided by the user. The transforms manipulate array statements in the code in such a way as to reduce the access overhead represented in the description. Dependence relationships obtained from earlier analyses between the statements are utilised in detecting constructs in the code that can be transformed.

Three transforms were presented which have been found to be applicable to the optimisation of examples for memory synthesis. Each seeks to reduce the memory access overhead for synthesised hardware by removing redundant accesses within the source code algorithm. Similar techniques have been reported elsewhere (see Section 2.2.1 and [18]) although these do not present automatic methods for optimisation.

Chapter 6

Evaluation of CSiC

In order to provide an evaluation of the CSiC tool, a pair of example applications are presented in this chapter. These have been selected so as to illustrate the structures present in typical application code that can be exploited for optimisation by the tool.

The first example is quite complex. It contains a large number of different arrays which are used throughout the code. It also contains a large number of loops, some with conditional constructs, making it a good example of a high-level image processing application.

The second example is more simple, containing a single loop at the top level. This is, however, heavily nested and a large number of array accesses take place within its body. This example has been implemented in hardware within Edinburgh University [27, 20] and so provides a useful benchmark for the CSiC system.

The example applications are presented as follows. A short explanation of each application is given, and an overview of the code for each is presented. This includes a review of the memory requirements of each example, as represented by the code. The results of running CSiC on the example are given. This comprises two parts, before and after optimisation. In each part the schedule results are presented for different architectures. The effect of memory organisation is analysed in each case, and the improvements gained by transforming the code descriptions is examined.

Each example is taken from a real application. My thanks go to Colin Ramsay

and Henry Bruce for providing me with the code samples used here.

6.1 Vector Quantization

In this example Vector Quantization [26], a signal processing technique used for the transmission of speech and image data, is utilised as a compression method for video signals. A typical application area for such a technique is video-phone transmission.

Compression is achieved by maintaining a codebook of data vectors at either end of the communications channel. The transmitter compares samples of data with vectors in the codebook and sends the index of the vector that gives the closest match. The receiver uses this index to reconstruct the original data from an identical codebook. This quantisation of an image into vectors makes this a lossy compression technique.

During image compression each vector represents a block of pixels in the original image, each with a different pattern of gray-scale values. For a block size of 4×2 pixels, 256 different vectors could be represented with a single byte, achieving a compression ratio of 8:1. The good performance is achieved at the cost of image quality as many blocks will be compressed to a closest match. Consequently the quality of the results is highly dependent upon the ability of the codebook to represent a wide enough range of vectors.

One way to minimise the quantization noise introduced is to use adaptive codebooks. These update their contents on the fly in an attempt to maintain good matching between image and codebook vectors. As the two sides' codebooks have to be identical, there must be some additional transmission in order to keep them synchronised. This degrades the compression ratio, but is acceptable when performed infrequently.

For transmission of image data where the scene is reasonably static quantization noise is less of a problem. In [112] the technique is applied to facial recognition, using feature analysis. Codebooks are constructed for facial features by training the system on a number of images of each subject. These codebooks can then be used as the basis for a facial recognition system.

The program used here is taken from a project at Edinburgh [113] which

constructs a codebook using a sequence of images. The code is an implementation of the Linde-Buzo-Gray (LBG) algorithm [26] for codebook design, shown in Figure 6.1.

```
Initialise the codebook
while (true) do
 ∀ vectors x_i in the training set do
  ∀ vectors v_j in the codebook do
   find the Euclidean distance, d_j, of x_i from v_j
  od
  find the minimum distance d_m
  assign x_i to the vector v_m
 od
 Replace each codeword with the centroid of
 the x vectors assigned to it
 Unused codewords in the codebook are replaced
 if Δ_error < some threshold then break
od
```

Figure 6.1. Outline of the LBG Algorithm

Initialisation of the codebook can be performed by a number of different methods, usually by selecting some population of vectors from the training set. Random selection of such vectors can be used. This example uses a sequence of images for training the codebook. Unused vectors are replaced with an imperfect copy of the most commonly used vector. The outer loop exit condition provides some measure of the changes made to the codebook compared with the previous iteration.

6.1.1 Example Overview

The program code used in this example performs codebook construction using a variant of the LBG algorithm. The top level procedure is shown in Figure 6.2. This implements a single phase of the LBG algorithm; quantization of the image, updating the codebook, then re-quantization of the image in order to check for a significant change in the codebook contents.

```
quantize_image(image,codebook,label_map);
update_codebook(codebook);
requantize_image(image,codebook,label_map);
```

Figure 6.2. Top Level Procedure for VQ Example

The procedure in Figure 6.2 is re-iterated until the sequence of images is complete. The images represent the training data and a single pass through them suffices for finding the end point of the LBG algorithm.

As will be seen much of the access activity occurs in the quantisation of the image. Each vector in the input image has to be compared with the vectors in the codebook in order to find the closest match. The worst case is that every image vector is compared with the entire codebook. In practice this is unlikely to happen, although CSiC assumes this worst case scenario. This results in much of the transformation effort being directed at this region of code and produces some difficulties when evaluating the effect of transforms applied to other parts of the algorithm.

The inline expansion of the procedures in Figure 6.2, and their subroutines, results in a program fragment consisting of 15 processes at the top level, in approximately 430 lines of C code, containing 68 symbols, amongst them 19 arrays. A summary of the arrays and their function within the program are shown in Table 6.1.

As the table shows, most of the memory space is the image array. The rest is concentrated in the various codebook structures. The number of different arrays of this type are an early indication that opportunities for access parallelism will exist in the example. Multiple arrays indicate a lot of intra-memory traffic and this increases the opportunities for exploitation of access redundancy, and therefore, optimisation of the description.

Structure	# Arrays	# Words	Comment
Codebook	9	15360	Storage of vectors, their usage, data for distance and splitting calculations
Image	1	65536	The training image store
Label Data	3	37280	Structures for holding data on vector labels
Ranking Data	6	3072	Arrays for data used in sorting b codewords
	19	111248	

Table 6.1. Arrays used in Vector Quantization Example

6.1.2 Process Scheduling

There are 45 loops in the program code and a total of 157 basic blocks. The process scheduling[1] finds a number of different global schedules for the program. The Parallel Access Matrix (PAM) for these processes is shown in Table 6.2. This contains the weights that indicate the access gain in having the process pairs execute in parallel.

Process Pair	Weight
133,134	98304
29,30	98304
64,74	2048
58,64	2044
58,74	2044
4,36	1024
36,71	512
4,71	512
88,89	447
97,98	96

Table 6.2. Parallel Access Weights for Top Level Processes

[1] see Section 4.1.1

When compared with the total access count for the algorithm these weights are small (see Table 6.6), indicating that the gains in memory throughput to be made by process parallelisation in this example are minimal. Table 6.3 shows the difference in the timing results for the two extreme process schedules (the maximally serial schedule and the maximally parallel schedule). The figures in the table differ by one hundredth of a percent, indicating any gain in using the parallel schedule will have negligible impact upon the hardwares performance. On the other hand there is no performance loss in choosing the maximally parallel schedule over the maximally serial either, nor is there any gain in cost. As mentioned earlier CSiC performs a conservative timing analysis, and with this example there is a data-dependent calculation to perform. With this knowledge, choosing the maximally parallel schedule might produce some performance payoff in the final solution.

Schedule	Time
Max Parallel	541659127
Max Serial	541719543

Table 6.3. Maximally Serial and Maximally Parallel Timing Results

The optimal schedule for the top level processes is shown in Figure 6.3. This schedule is built using the weights of Table 6.2 and the inter-process dependences. The tool has been able to use all the parallel weights to construct the maximally parallel schedule. The processes that are parallelised mainly perform the initialisation of various arrays in the code. Their access weights are added into the global access matrices. This will have an effect upon the allocation of the arrays into different memories at logical allocation time. The intention here is to favour memory organisations that take advantage of the parallelism in the algorithm by separating data that can be accessed in parallel into separate memories.

6.1.3 Initial Synthesis

The example contains 35 basic blocks which access array memory. The first

Level	Step	Processes
top	1	71, 36, 4
	2	13
	3	40
	4	74, 64, 58
	5	65
	6	82
	7	86
	8	95
	9	108
	10	117
86	1	156
	2	89, 88
95	1	94
	2	98, 97
16	1	19
	2	30, 29
120	1	123
	2	134, 133

Figure 6.3. Maximally Parallel Schedule for Processes

stage of the synthesis procedure is the maximal scheduling[2] of these blocks individually. Once this is accomplished then logical allocation can begin.

The graph in Figure 6.4 shows the optimal memory schedule lengths for the blocks in the program. These are the minimum schedules allowed by the inter-access dependences, and represent the maximally parallel schedules for all blocks. The bar graph gives the block count for schedules of different lengths in the program. The width of the bars is proportional to the amount of time spent in the blocks having that length. This has been calculated from the number of times each block is executed within the program.

From the graph we see that the spread of schedule lengths is small, indicating their optimal length. The tool aims to synthesise architectures that support schedules as close to this scheme as possible. Further transformation of the input

[2]see Section 4.1.2

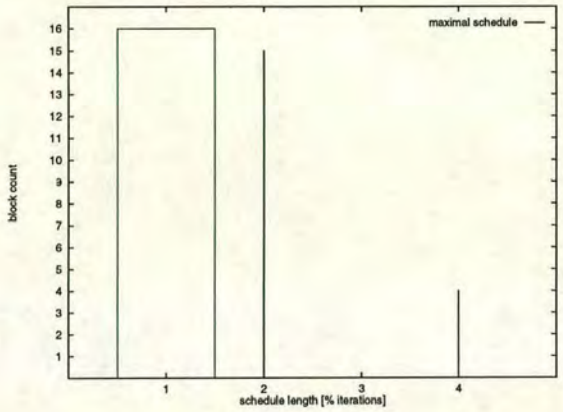

Figure 6.4. Optimal Schedule Lengths for Basic Blocks

description will aim to increase this optimality further. From the graph we see that $\frac{16}{35}$ of the block schedules have an optimal schedule length of 1, and that these are responsible for most of the accessing within the algorithm.

Once the optimal schedule is created the Block Allocation Matrices (BAMs) are generated. These are used for allocating arrays to memories within the different architectures[3]. Part of the parallel BAM is shown in Table 6.4.

From the table we see that a small number of array pairs have significantly more parallel access potential than others. The memory allocation process concentrates on attempting to place these array pairs into separate memories.

The allocation procedure iteratively generates architectures for the description. Each architecture is assigned a number of memories and the allocator places arrays into these memories according to the weights in the BAMs. The number of architectures generated by CSiC is, by default, equal to the number of arrays in the description. Once this is accomplished an updated schedule is created for each architecture. This final schedule provides the basis for the evaluation of the completed architectures.

The allocation achieved for the BAM of Table 6.4 is shown in Table 6.5. From this table we see that the allocation algorithm has succeeded in placing the most significantly weighted array pairs into separate memories wherever possible. This will ensure that the generated schedules will be able to maximally exploit

[3]see Section 4.1.5

Array Pair	Parallel Weight	Array Pair	Parallel Weight
198,172	134217728	176,175	521220
192,193	521220	219,218	521220
173,198	262144	165,173	32768
165,198	32768	166,165	32768
166,173	32768	166,198	32768
218,176	16896	219,176	16896
165,201	16384	166,201	16384
169,165	16384	169,166	16384
169,173	16384	169,198	16384
169,201	16384	173,201	16384
198,201	16384	218,174	16384
219,174	16384	166,219	9216
166,176	8704	166,218	8704
166,172	8192	172,176	8192
172,218	8192	172,219	8192
173,172	8192	173,176	8192
173,218	8192	173,219	8192

Table 6.4. Most Significant Arrays Pairs

the access parallelism present within the basic blocks of the application code.

In practice not all the architectures are optimal. Once a certain number of memories have been allocated the updated schedules exploit all the available parallelism in the blocks and adding further memories only increases the cost of the solution, with no further performance gains. Additionally, if there is no potential parallelism in those blocks where most of the access activity takes place, allocating extra memories will have a negligible effect on the overall performance.

This is illustrated in Figure 6.5. The graphs in this figure show the effect of increasing the memory count upon the overall schedule lengths. The width of the bars in the graphs give the proportion of accesses performed in blocks of the indicated length.

As the number of memories in the architecture increases the graphs tend towards the optimal schedule graph of Figure 6.4. The optimal schedule profile is reached for a seven memory architecture, however the difference in performance

Arch	Mem #	Allocation
1	1	166 167 171 174 176 192 198 201 218 224
	2	165 169 172 173 175 193 219
2	1	165 167 171 174 176 192 198
	2	166 172 175 201 218 224
	3	169 173 193 219
3	1	167 171 192 198 201 219 224
	2	172 173 174 175
	3	165 169 193 218
	4	166 176
4	1	167 171 192 198 218
	2	166 172 174 175
	3	169 173 193
	4	165 176
	5	201 219 224
5	1	167 171 192 198
	2	166 172 174 175
	3	173 193
	4	165 176
	5	201 219 224
	6	169 218

Table 6.5. Allocations for Example

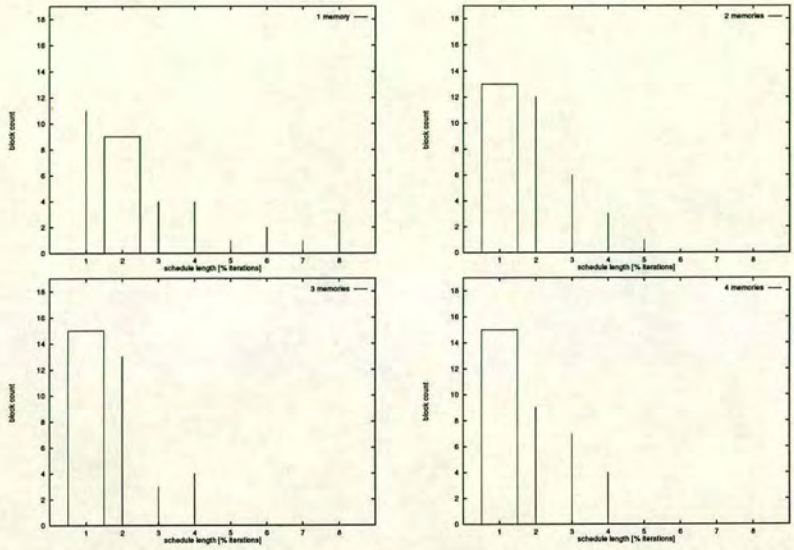

Figure 6.5. Block Schedule Lengths for Different Architectures

between the two memory and the optimal architecture is negligible.

# Memories	Time	Cost	PI	CI
1	541659127	131072	1.00	0.00
2	271133675	163840	2.00	0.25
3	271132619	180224	2.00	0.38
4	271133643	166912	2.00	0.27
5	271132139	214016	2.00	0.63
6	271132107	215040	2.00	0.64

Table 6.6. Timing Data for the Architectures of Figure 6.10

From these graphs it can be seen that increasing the memory count beyond two memories will have little appreciable effect on the performance of the synthesised solutions. The bulk of the time spent is in accessing between arrays 198 and 172 (from Table 6.4) and it is this activity which results in the large bar in Figure 6.5. This is borne out by the timing data presented in Table 6.6.

The table shows the access count for each architecture, calculated from the generated schedules. The total memory allocated for each architecture is also

shown, as a rough measure of cost. The fourth column in the table shows the performance increase (PI) for each architecture over the single memory solution. A similar measure is given for cost increase (CI) in the fifth column of the table.

As shown by the schedule graphs above, an appreciable performance gain is only achieved by incrementing the number of memory units to two. This is due to the access bottleneck that exists in the algorithmic code whereby the bulk of the accessing is performed in two blocks. The PI column of Table 6.6 illustrates this.

When allocating memory CSiC minimises physical memory by allocating memory blocks to the allocated arrays with sizes that factor by two. The tool attempts to avoid fragmentation within memory by placing non-critical arrays into spaces that will not increase the eventual physical memory size. This is, however, a secondary consideration compared with obeying the significant weighted pairing given in the BAM.

The difference in the memory costs of the three and four memory solutions in Table 6.6 is accounted for by a decrease in fragmentation within the memories, allowing a smaller amount of physical memory to be allocated.

Synthesis Summary

From the results presented above it is clear that the two memory solution provides the largest increase in performance, doubling the access bandwidth between the processing core and the memory. This is achieved by an increase in memory cost of 25%, from 132 kbytes to 164 kbytes. Increasing the number of memories further has no appreciable impact on the performance. This is due to an access bottleneck during the quantisation stage of the algorithm.

In order to improve on the synthesis it is necessary to try to break this bottleneck. This is achieved by transformation of the source code and repeating the synthesis procedure. The next section describes this process for the VQ example.

6.1.4 Applying the Transforms

As described in Chapter 5 the opportunity to perform a particular transform is detected by analysing the dependences between array statements in the code

description. Table 6.7 shows the relevant simple dependences detected in the code for the VQ example, these indicate statements that can be transformed using the SRR Transform. Table 6.8 shows clusters of carried dependences between statements in the code. These indicate sites of potential application of the CRR Transform.

Cluster	Dependences
5	$S_{402}\ \delta^{i}_{00}\ S_{410}$
6	$S_{371}\ \delta^{i}_{00}\ S_{360}\quad S_{371}\ \delta^{*}_{00}\ S_{366}\quad S_{360}\ \delta^{*}_{00}\ S_{366}$
8	$S_{369}\ \delta^{i}_{00}\ S_{148}\quad S_{369}\ \delta^{*}_{00}\ S_{361}\quad S_{148}\ \delta^{*}_{00}\ S_{361}$
9	$S_{353}\ \delta^{*}_{000}\ S_{358}\quad S_{133}\ \delta^{o}_{0}\ S_{358}\quad S_{133}\ \delta^{f}_{0}\ S_{353}$
10	$S_{348}\ \delta^{f}_{000}\ S_{354}$
11	$S_{326}\ \delta^{*}_{00}\ S_{332}\quad S_{337}\ \delta^{i}_{00}\ S_{326}\quad S_{337}\ \delta^{*}_{00}\ S_{332}$
12	$S_{335}\ \delta^{i}_{00}\ S_{115}\quad S_{335}\ \delta^{*}_{00}\ S_{327}\quad S_{115}\ \delta^{*}_{00}\ S_{327}$
14	$S_{274}\ \delta^{*}_{0}\ S_{276}\quad S_{91}\ \delta^{*}_{0}\ S_{276}\quad S_{91}\ \delta^{i}_{0}\ S_{274}$
15	$S_{299}\ \delta^{*}_{00}\ S_{305}\quad S_{310}\ \delta^{i}_{00}\ S_{299}\quad S_{310}\ \delta^{*}_{00}\ S_{305}$
16	$S_{277}\ \delta^{i}_{0}\ S_{286}$
18	$S_{308}\ \delta^{i}_{00}\ S_{96}\quad S_{308}\ \delta^{*}_{00}\ S_{300}\quad S_{96}\ \delta^{*}_{00}\ S_{300}$
20	$S_{255}\ \delta^{i}_{00}\ S_{263}$
22	$S_{143}\ \delta^{i}_{0}\ S_{350}$
24	$S_{89}\ \delta^{f}_{0}\ S_{293}$

Table 6.7. SRR Transform Dependences for the VQ Example

The clusters are those that CSiC has extracted from the dependence analysis. Many of the generated clusters are useless for the purposes of transformation. This is because the tool does exhaustive testing for dependences between access statements within loops and consequently detects many sites where transformation cannot occur. The algorithm for extracting useful dependences from the sets produced is given in Section 5.5.

Because of the inlined nature of the code[4] there is also some repetition in the dependences. The code essentially has three parts at the top level (see Figure 6.2). The first and third parts are the same and so the same dependences will exist

[4]see Section 6.1.1

within them. These parts also contain the main bottlenecks in the code sample, as this is where the image quantisation takes place.

Cluster	Dependences
3	$S_{403}\ \delta^{i}_{-1-1000}\ S_{403}$
7	$S_{402}\ \delta^{i}_{00-100}\ S_{402}$
14	$S_{358}\ \delta^{\circ}_{0-10}\ S_{358}\quad S_{358}\ \delta^{f}_{0-10}\ S_{353}\quad S_{353}\ \delta^{*}_{0-10}\ S_{358}$
21	$S_{256}\ \delta^{i}_{-1-1000}\ S_{256}$
31	$S_{350}\ \delta^{i}_{0-10}\ S_{350}$
36	$S_{255}\ \delta^{i}_{00-100}\ S_{255}$

Table 6.8. CRR Transform Dependences for the VQ Example

From tables 6.7 and 6.8 we see that there are some dependences that affect the same statements in each table. These are clusters (5,7), (9,14) and (20,36) of the SRR and CRR sets respectively. The (5,7) and (20,36) cluster pairs are the same, each occurring in the same piece of inlined code. The transformation of the code affected by these dependence pairs illustrates the all three types of dependence transformation: Simple and Carried Redundance Removal, and Access Motion. The cases for dependence pairs (5,7) and (9,14) are given below and illustrate the transformational stage of the CSiC synthesis strategy.

Access Motion and Simple Redundancy Removal

The dependence clusters (5,7) in tables 6.7 and 6.8 are

$$S_{402}\ \delta\ ^{i}_{00}\ S_{410}$$

and

$$S_{402}\ \delta\ ^{i}_{00-100}\ S_{402}$$

The first is an uncarried (or simple) dependence, the second a loop carried dependence. The second shows an input dependence that is carried by a single iteration of the level 3 loop, enclosing statement 402, which is dependent upon itself. This indicates that an Access Motion Transform[5] can be applied to the source code, moving the statement from level 5 to level 2. The transform requires that a shift register be introduced to the source, with a size of $I_4 \times I_5$ where I_n refers to the iteration count of the loop at level n.

The first dependence, also an input dependence, shows that when $i_4 = j_4$ and $i_5 = j_5$, where i_n and j_n are the loop induction variables for statements 402 and 410 at levels n, respectively, that the same data is read from the array. This indicates redundant accesses are taking place. The redundancy can be removed by accessing a local variable with the appropriate value, in this case, those held in the shift register introduced by the Access Motion transform. The source code for both these dependences is shown in Figure 6.6.

In this code fragment we see the loops at levels 3 to 5 around statement 402. The variables `_tmp147` and `_tmp146` are initialised at level 3. These provide an offset into the input image for a block that is to be quantized. The same offset is used in both loops (40,41) and loops (42,43) (the third comment field in the CSiC -generated code gives the loop number, the second is the statement number and the first is the basic block number). The result of the two transforms is shown in Figure 6.7.

The access reduction obtained from these transforms is as follows. For the

[5]see Section 5.4

Access Motion transformation, we lose

$$\prod_{k=1}^{k=j} I_k(\prod_{k=i-j}^{k=i} I_k - 1)$$

where j is the level of the leftmost '-1' in the distance vector of the dependence, i is the innermost level. The outer level iterations ($k = 1$ and $k = 2$) are each 128, so the access reduction is $128 \times 128 \times 512 \times (4 \times 4 - 1) = 125829120$. Such a large reduction in the access count (94%) is not unusual when taking an algorithmic description from C code to hardware. The original description was not optimised for access, the code expressed only the algorithm required to perform the task. In the original description the inner loop was executed within a procedure call. The procedural inlining of the code has enabled CSiC to spot this opportunity for reducing the access count as the inlining has flattened the programmer's code hierarchy.

In the case of the SRR Transform a further $128 \times 128 = 16384$ accesses will not appear in the transformed source, due to the re-use of the (circular) shift register. This has little further impact than the reduction gained from the Access Motion transform.

```
    ...
_tmp127=0; /* 118,225,38 */
for (;_tmp127<512;) /* 123,223,39 */ {
  _tmp131=0; /* 121,218,39 */
  for (;_tmp131<4;) /* 126,216,40 */ {
    _tmp132=0; /* 124,215,40 */
    for (;_tmp132<4;) /* 129,213,41 */ {
      _tmp104=_tmp147+_tmp132; /* 128,400,41 */
      _tmp105=_tmp146+_tmp131; /* 128,401,41 */
      _tmp106=input_image[_tmp105][_tmp104]; /* 128,402,41 */
          ...
    }
    ++_tmp131; /* 127,217,40 */
  }
      ...
}
_tmp123=0; /* 122,232,38 */
for (;_tmp123<4;) /* 134,230,42 */ {
  _tmp126=0; /* 132,229,42 */
  for (;_tmp126<4;) /* 137,227,43 */ {
        ...
    _tmp111=_tmp147+_tmp126; /* 136,408,43 */
    _tmp112=_tmp146+_tmp123; /* 136,409,43 */
    _tmp113=input_image[_tmp112][_tmp111]; /* 136,410,43 */
...
  }
  ++_tmp123; /* 135,231,42 */
}
```

Figure 6.6. Original Code for AM and SRR Example

```
    ...
for (_tt1 = 0;_tt1 < 4;_tt1++) {
  for (_tt2 = 0;_tt2 < 4;_tt2++) {
    _tmp104=_tmp147+_tt2;_tmp105=_tmp146+_tt1;
    _shift16 = input_image[_tmp105][_tmp104];
  }
}
_tmp127=0; /* 118,225,38 */
for (;_tmp127<512;) /* 123,223,39 */ {
  _tmp131=0; /* 121,218,39 */
  for (;_tmp131<4;) /* 126,216,40 */ {
    _tmp132=0; /* 124,215,40 */
    for (;_tmp132<4;) /* 129,213,41 */ {
      _tmp106 = _shift16;
          ...
    }
    ++_tmp131; /* 127,217,40 */
  }
      ...
}
_tmp123=0; /* 122,232,38 */
for (;_tmp123<4;) /* 134,230,42 */ {
  _tmp126=0; /* 132,229,42 */
  for (;_tmp126<4;) /* 137,227,43 */ {
        ...
    _tmp113 = _shift16;
...
  }
  ++_tmp123; /* 135,231,42 */
}
```

Figure 6.7. Transformed Code for AM and SRR Example

Carried Redundancy Removal

The second cluster of dependences, (9,14) in the tables above, are

$$S_{353}\ \delta\ {}^{*}_{000}\ S_{358} \quad S_{133}\ \delta\ {}^{\circ}_{0}\ S_{358} \quad S_{133}\ \delta\ {}^{f}_{0}\ S_{353}$$

for the simple dependences, and

$$S_{358}\ \delta\ {}^{\circ}_{0-10}\ S_{358} \quad S_{358}\ \delta\ {}^{f}_{0-10}\ S_{353} \quad S_{353}\ \delta\ {}^{*}_{0-10}\ S_{358}$$

for the carried dependences. The code for these dependences is shown in Figure 6.8.

```
k=0; /* 70,146,0 */
for (;k<512;) /* 74,144,23 */ {
  centring_change_r_meas[k]=0; /* 72,133,23 */
  if (cb_no_assigned[k]) /* 72, 143,23 */ {
    _tmp139=0; /* 75,142,23 */
    for (;_tmp139<4;) /* 78,140,24 */ {
      j=0; /* 77,139,24 */
      for (;j<4;) /* 81,137,25 */ {
      ...
          _tmp72=centring_change_r_meas[k]; /* 80,353,25 */
      ...
          centring_change_r_meas[k]=_tmp76; /* 80,358,25 */
          ++j; /* 80,138,25 */
      }
      ++_tmp139; /* 79,141,24 */
    }
  }
  ++k; /* 76,145,23 */
}
```

Figure 6.8. Pre-transformed Code for the CRR Example

A number of dependences affect the three statements and taken together these allow two transforms to take place. The carried output dependence $S_{358}\ \delta\ {}^{\circ}_{0-10}\ S_{358}$ on statement 358 allows an Access Motion transform to be

applied. This moves statement 358 to the end of the two nested loops, removing the write redundancy that exists in the untransformed code. A temporary variable is introduced which carries the value of the memory location during the loops' lifetimes. This scalar value replaces the references to the memory location within the loops' body, further reducing the access count for the fragment.

```
k=0; /* 70,146,0 */
for (;k<512;) /* 74,144,23 */ {
  tt3 = 0;
  if (cb_no_assigned[k]) /* 72, 143,23 */ {
    _tmp139=0; /* 75,142,23 */
    for (;_tmp139<4;) /* 78,140,24 */ {
      j=0; /* 77,139,24 */
      for (;j<4;) /* 81,137,25 */ {
    ...
        _tmp72 = tt3;
    ...
        tt3 = _tmp76; /* 80,358,25 */
        ++j; /* 80,138,25 */
      }
      ++_tmp139; /* 79,141,24 */
    }
  }
  centring_change_r_meas[k] = tt3;
  ++k; /* 76,145,23 */
}
```

Figure 6.9. Transformed Code for CRR Example

Additionally the uncarried dependence between statements 133 and 358 allows the write of statement 133 to be removed also. Note that this means statement 358 must be moved to outside the conditional statement number 143 to ensure that the code remains semantically correct. The transformed code fragment is shown in Figure 6.9.

This method is an extension of the algorithm described in Section 5.5, which currently handles dependences of the same type. However, it is a fairly trivial extension, only requiring that the input dependence set contains all the relevant

dependences for the statements, and that the if-statement is accommodated.

The access reduction for this example is

$$512 \times 16 \times 2/(512 + 512 \times 16 \times 2) = 97\%$$

This shows the potential of the transforms when applied to users code. Whilst the impact on the total count may not be high for this particular example, it serves to illustrate the power of dependence analysis to optimise users code for further synthesis.

Further Transformation

One other carried dependence provides the user with a significant design decision. This is again duplicated in the image quantisation sections of the code and would require the addition of an 8 kilobyte on-chip memory to the design. The data that would reside in the memory would be the code book vectors. These would have to be loaded on to the chip at the start of the process (and presumably written out at the end). Alternatively the vectors could start initialised to some, possibly random, value and the entire algorithm be allowed to iterate until a steady state is reached.

This would gain the user a large performance gain, but at the cost of a large integrated memory. The dependences are clusters 3 and 21 in Table 6.8, each indicates a carried dependence across the outer two loops. These loops execute their inner body 16384 times, and CSiC 's conservative timing shows that each code book vector is accessed during each of these iterations.

At present CSiC does not support the inclusion of on-chip memory blocks into the synthesised architectures, so it is not possible to obtain results for this particular transform. Removing accesses to the code book gives a simulation of the timing results that would be obtained by such an addition to the architecture and these are included in the next section.

The results of the above transformations are presented below, followed by a further set which result from the application of this last transform to show the full potential of the CSiC -generated architectures. The remaining SRR and AM dependences allow relatively trivial transforms to be applied to the source. They

are a mixture of SRR and AM transforms that remove access redundencies from the code by scalar substitution.

Transformed Results

The final schedules for the various architectures are presented in Figure 6.10. The application of the transforms described above result in an increase in the performance of the single memory architecture similar to that obtained for the two-memory architecture synthesised from the un-transformed code. This is due to the introduction of the shift register for the current image block which halves the main access bottleneck in the program.

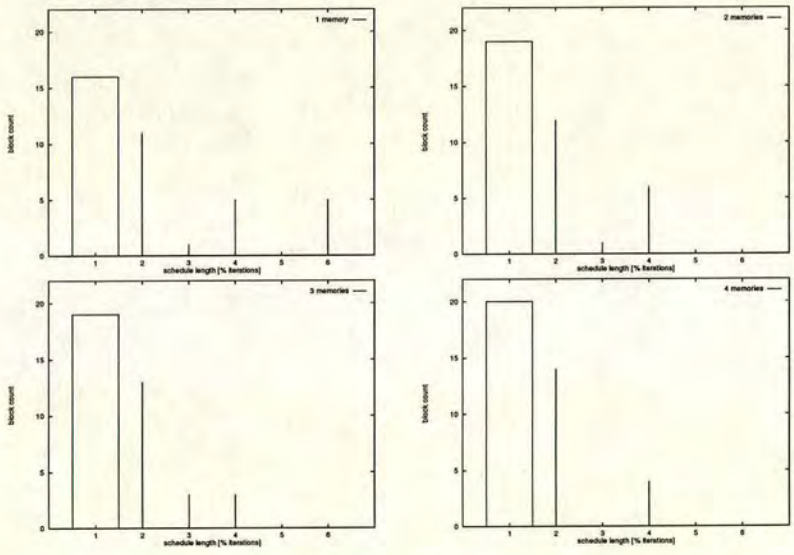

Figure 6.10. Block Schedule Lengths for Different Architectures

These graphs show that the access bottleneck has been reduced by transforming the source code. Looking at the single memory solution we see that the bulk of the accessing is performed in schedules with a length of a single access. The schedule has achieved the same number of single length steps as that obtained by the untransformed optimal schedule. The effect on the timing for the algorithm can be seen from the data for the architectures in Table 6.9.

For architectures with two or more memories the schedule profiles overtake

the ideal (maximally parallel) schedule profile. This illustrates the optimising effect of the transforms on the user code. This suggests that the application of the transforms has a similar effect to increasing the memory count, and that both techniques produce more optimal results for the synthesised algorithm.

# Memories	Time	Cost	PI	CI
1	271753201	131072	1.00	0.00
2	270935499	151552	1.00	0.16
3	270919115	149504	1.00	0.14
4	270902219	146432	1.00	0.12
5	270902251	160256	1.00	0.22

Table 6.9. Timing Data for the Transformed Schedules

The graphs in Figure 6.11 simulate the effect of adding an 8 kb on-chip memory to the architecture to hold the vectors for the quantised image. This effectively removes the need to access the vectors at all from external store and greatly increases the algorithm's performance. The timing for the architectures can be seen in Table 6.10. Here we see a performance improvement of 7000% for the single memory architecture with the extra memory.

The costing data of Table 6.10 does not include this extra on-chip cost, naively it would represent a 6% increase over the figures in the table. However this does not take into account the extra costs incurred in having memory on-chip as this is outside of the scope of the current costing strategy.

The schedule profiles for this architecture do not resemble the optimal schedule atall. This is due to the withdrawal of a number of accesses from the algorithm, breaking the bottleneck between the vector and image memories.

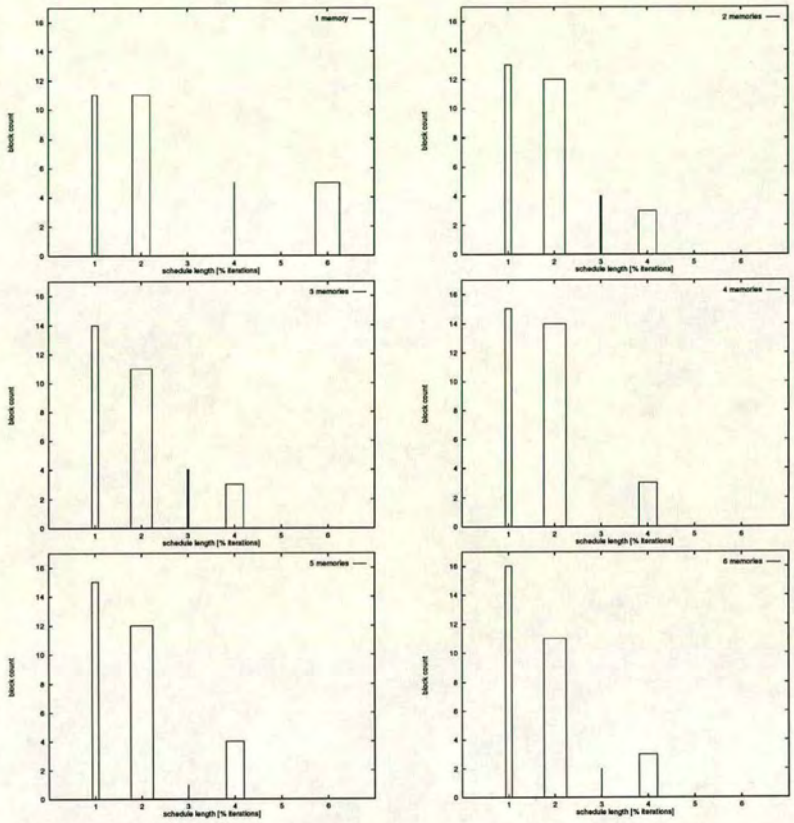

Figure 6.11. Synthesis Results for Example with Extra On-chip Memory

# Memories	Time	Cost	PI	CI
1	3809409	131072	1.00	0.00
2	2976283	147456	1.28	0.12
3	2976315	150528	1.28	0.15
4	2958875	145408	1.29	0.11
5	2958459	152064	1.29	0.16

Table 6.10. Timing Data for Example with Extra On-chip Memory

6.2 Finger Print Verification

The example presented in this section provides a useful case for the evaluation of CSiC . The algorithm has been implemented as an ASIC design [20], which was a result of research conducted at the University of Edinburgh into finger print verification using template matching techniques [27]. The core of the algorithm involves an adaptively thresholded fingerprint image being presented for correlation with a database of templates. It is this core which is used as the example here. The low level processing of the grey scale image is omitted, it is assumed that this has been performed at an earlier stage.

Similarly the template generation is not dealt with. Both of these procedures are also heavily constrained by memory access, although in the latter case the procedure is not so time critical and template updating and could be performed off-line. The part of the example given here, then, deals exclusively with the actual comparison executed by the algorithm.

6.2.1 Example Overview

The main algorithm is heavily constrained by memory access. It is composed of a single process which comprises six tightly nested loops. The outer loop iterates through the template database until a match is found. The inner loops perform the comparison between the presented binarised image and these templates. These loops' iterations are summarised as follows:-

- Iterate through different angles to allow for variations in the presentation of the input image.
- Iterate through a number of blocks in the input image.
- Correlate each block with a template, counting mismatches between pixels. Go to the next block if the number of mismatches exceeds a threshold.
- If the correlation is successful then indicate a verification has been made.

The example used here does not include the conditional branches for breaking out of the comparison. Such tests are implemented in the hardware control

logic and can be ignored for the memory architecture. Further development of CSiC would involve the synthesis of control logic for datapath, address generation and the memory interface where these issues would be dealt with.

There are nine arrays declared in the compare algorithm. The array sizes are shown in Table 6.11. Most of the data is held in the image array, the rest is in relatively smaller structures. The numbers in parenthesis will be used in later tables to refer to the arrays.

Array Name	Dimensions	Size (bytes)	Remarks
image (46)	2	65536	The smoothed and binarised input imag
signature (60)	3	1152	An array of signatures to be compared
roffset (55)	3	2688	Row offset for different angles
coffset (36)	3	2688	Column offsets for different angles
sig_pts (59)	1	9	
base_sig_pos_row	1	9	
base_sig_pos_col	1	9	
rownum (57)	1	16	
colnum (38)	2	128	
		72235	

Table 6.11. Arrays in the Compare Algorithm

The program consists of a single procedure which performs a comparison of a single binarised fingerprint image against a database of fingerprint signatures. The algorithm allows for variation in the presentation of the fingerprint by performing tests through a variety of different angles. The development of the algorithm is covered in [27]. A hardware implementation of the algorithm is given in [20]. This implementation is compared with CSiC 's output in Section 6.2.5.

6.2.2 Process Scheduling

There are six tightly nested loops in the program, all enclosing a conditional test. This means that CSiC finds only a single global process schedule for the algorithm as there is no possiblity of parallelising any of the basic blocks. There are seven of these which contain array access statements in the description. The body of

the conditional statement contains array accesses; as in the previous example, CSiC assumes the worst case timing in this instance, that is that all conditional branches will be taken.

This means that access counts for statements in the innermost loop are essentially worst case. The curent implementation of CSiC does not make any special allowance for this situation, the tool only assumes that that potential access count exists and has to be minimised as best as possible.

6.2.3 The First Iteration

Array Pair	Parallel Weight	Array Pair	Parallel Weight
36,55	480961152	36,60	480961152
38,36	480961152	38,55	480961152
38,60	480961152	55,60	480961152

Table 6.12. Parallel Access Weights for Arrays

The parallel weights for the different arrays in the code are shown in Table 6.12. The matrix includes only four of the arrays in the program. This suggests that a large numbr of memories will not necessarily benefit the synthesis results. The column and row offset arrays and the signature array (nos. 36, 55 and 60) are particularly important in this respect. These arrays will play a significant role in the program's access bottlenecks.

These weights originate from the statements in the inner block, the code for which is shown in Figure 6.12. Statements 47, 42, 44 and 50 can all be executed in parallel as no dependences exist between them. CSiC clusters these statements in the maximal parallel schedule from which the access weights are drawn.

The image array (46) access in statement 51 could potentially parallelise with the signature (60) acccess of statement 50. This would lead to an entry in the BAM for the array pair (46,60). Because the algorithm that produces the ideal schedules[6] always aims to obtain maximal parallelism, statement 50 is moved to the same schedule step as the maximum number of other statements that the

[6]See Section 4.1.2

```
_tmp7=roffset[abs_angle][rn][cn]; /* 20,47,6 */
_tmp8=row+_tmp7; /* 20,48,6 */
r=_tmp8; /* 20,49,6 */
_tmp3=colnum[rn][cn]; /* 20,42,6 */
_tmp4=first_col+_tmp3; /* 20,43,6 */
_tmp5=coffset[abs_angle][rn][cn]; /* 20,44,6 */
_tmp6=_tmp4+_tmp5; /* 20,45,6 */
c=_tmp6; /* 20,46,6 */
_tmp9=signature[s][rn][cn]; /* 20,50,6 */
_tmp10=image[r][c]; /* 20,51,6 */
_tmp11=_tmp9-_tmp10; /* 20,52,6 */
```

Figure 6.12. Code for Inner Block of Compare Example

data dependences will allow. This has the potential to affect the optimality of schedules for low-memory count architectures as the array pair (46,60) could be allocated to the same memory.

The allocation of the arrays into memory is in Table 6.13. Here we see that in each architecture the array pair (46,60) are always in different memories, despite their not having an entry in the BAM. From Section 4.1.3 two types of BAM are generated, a serial access matrix and a parallel access matrix. It is the Parallel Access Matrix (PAM) that is in Table 6.12. From Figure 6.12 can be seen that the serial BAM will contain entries for the image array and all the other arrays in the block (as data dependences exist between them), but not between the image and signature arrays. The algorithm described in Section 4.1.5 attempts first to allocate all array pairs in PAMs into separate arrays, then all array pairs in SAMs into the same arrays.

This will mean that all the potential parallelism in the block can be exploited when the schedules come to be generated, statements 50 and 51 will be able to execute in parallel through the memory architecture.

The allocations of Table 6.13 are for architectures with two or more memories,

Arch	Mem #	Allocation
2	1	31 32 55 57 59 60
	2	36 38 46
3	1	36 55
	2	38 46
	3	31 32 57 59 60
4	1	55
	2	38 46
	3	31 32 57 59 60
	4	36
5	1	55
	2	38 46
	3	31 32 57 59 60
	4	36
	5	

Table 6.13. Array Allocation

the single memory case will contain all arrays. Allocations up to only five memories are shown, as increasing the memory count any further has no additional benefit. The algorithm will only place arrays in separate memories if there is an entry in either BAM for that array pair, as illustrated in the case of arrays 31, 32, 57, 59 and 60.

Synthesis Summary

Once the allocation has been performed, CSiC generates the schedules for each architecture. These are shown in Figure 6.13 for architectures of up to six memories. As with the previous example, the effect of adding more memories to the architecture is to reduce the length of the schedules. In this case no schedule lengths can be reduced to a single step, parallelisation is gained in the longer schedules.

The two and three memory schedule profiles are the same, it is not until a fourth memory is added to the architecture that a further improvement in performance is achieved. This is due to a weakness in the block scheduling algorithm which does not take account of the potential of re-ordering the statements within

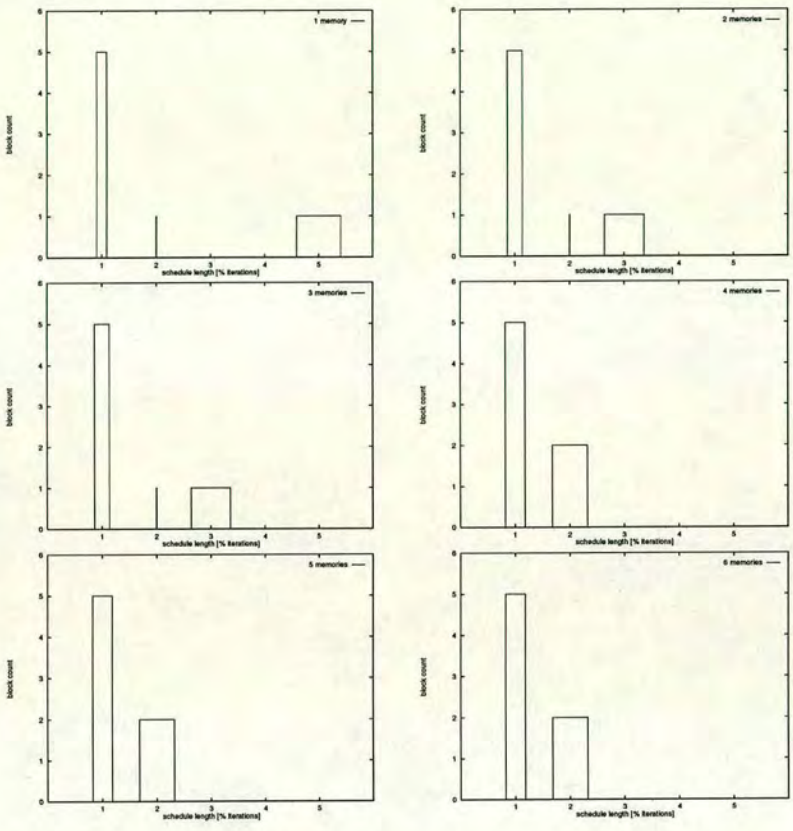

Figure 6.13. Block Schedules for Different Architectures

the schedules to take advantage of the extra parallelism afforded by statements 50 and 51. The schedules for the two, three and four memory architectures, for block 20, are shown in Figures 6.14, 6.15 and 6.16.

Whilst the two and four memory schedules maximise their use of the parallel potential in the basic block, the three memory schedule has three steps when it only really needs two. The signature and image array loads could occur in parallel, whilst the roffset load in step 2 could move into step 1. In order for the CSiC scheduling algorithm to achieve this it would have to do a list ordering on the statements taken from the maximal schedule steps in order to decide which statement to attempt to schedule next. At present the algorithm takes an arbitrary ordering from each step. One method of ordering the list would be to perform a force-directed type scheduling of the list (see Section 2.1.3). Another

Step	Arrays
1	load(colnum) load(roffset)
2	load(coffset) load(signature)
3	load(image)

Figure 6.14. 2 Memory Schedule

Step	Arrays
1	load(colnum) load(coffset) load(signature)
2	load(roffset)
3	load(image)

Figure 6.15. 3 Memory Schedule

Step	Arrays
1	load(colnum) load(coffset) load(roffset) load(signature)
2	load(image)

Figure 6.16. 4 Memory Schedule

method of prioritising the list would be to take an immediacy-type measure for each statement to be scheduled. Statements that could be parallelised with others further down the list would have a lower prioity than those with fewer potential parallelisations.

The timing and cost results for the architectures are shown in Table 6.14. These reflect the missed opportunity for optimising the three memory schedule further, as the timing result is the same for the two and three memory architectures. A slight reduction in cost is obtained moving from two to three memories, indicating a reduction in fragmentation within the memory blocks. As was detailed in Section 4.3, the costing performed by CSiC is quite simple, no allowance is made for the increase in addressing hardware required to support the extra memory. This issue has been the subject of recent research (see [107]) and its significance for CSiC requires further investigation.

# Memories	Time	Cost	PI	CI
1	2476225089	73728	1.00	0.00
2	1514302785	73728	1.64	0.00
3	1514302785	73088	1.64	-0.01
4	1033341633	73088	2.40	-0.01
5	1033341633	73088	2.40	-0.01
6	1033341633	73088	2.40	-0.01
7	1033341633	73088	2.40	-0.01
9	1033341633	73088	2.40	-0.01

Table 6.14. Timing and Cost Data for the Example

6.2.4 Transforming the Description

In order to improve the synthesis of the user description it is necessary to transform the algorithm. This is detailed below as a two stage process. First the so-called simple transforms are covered. These introduce no extra memory cost to the final solution, although additional local variables are introduced which may have an impact on register allocation in the functional solution. The current implementation of CSiC does not count such costs.

Simple Transformations

Cluster	Dependences
2	$S_{54}\ \delta^{i}_{000000}\ S_{50}$
59	$S_{59}\ \delta^{i}_{0000}\ S_{62}\quad S_{68}\ \delta^{i}_{0}\ S_{62}\quad S_{68}\ \delta^{i}_{0}\ S_{59}$

Table 6.15. The Program Dependences

The first set of transforms applied to the code are the so-called Simple Transforms (SRRT). These look for loop independent relations between access statements in the code. They result in a redundancy removal involving the insertion of a single scalar variable into the code. The clusters of simple dependences found by CSiC are summarised in Table 6.15.

```
    ...
_tmp12=signature[s][rn][cn]; /* 17,54,6 */
if (_tmp12!=2) /* 17, 55,6 */
{
      ...
  _tmp9=signature[s][rn][cn]; /* 20,50,6 */
      ...
}
```

Figure 6.17. Example Code

As can be seen from the table the main saver is a dependence between two statements in the innermost loop. The statement S_{50} reads a value from memory

that has already been read by statement S_{54}. The target statement in the dependence is situated in the conditional block and so the saving estimate will never be as large as that given in the table. However, the size of the saving indicates the importance of the transform for the memory organisation's efficiency. The code in Figure 6.17 shows the site of the dependence before the application of the transform. Figure 6.18 shows the code after transformation by the SRRT.

```
    ...
_tmp12=signature[s][rn][cn]; /* 17,54,6 */
t1 = _tmp12;
if (_tmp12!=2) /* 17, 55,6 */
{
      ...
  _tmp9=t1; /* 20,50,6 */
      ...
}
```

Figure 6.18. Example Code

The other two dependences in Table 6.15 indicate that statement 68 has an input dependence with statements 59 and 62. The application of the SRRT results in a temporary variable replacing the access to the array in both of these statements. A dependence also exists between statements 62 and 59, which reside in the same basic block. Statement 68 comes before both statements, in a different block. The dependence clustering and statement dominance analysis performed by CSiC supplies the dependences that will result in the optimal transform being applied.

The timing results for the application of the transforms are shown in Table 6.16. These show a performance improvement of 25%, <1% and 48% for the one, two and three memory architectures respectively. The large gain for the three memory architecture is explained by the sub-optimal solution obtained for the untransformed three memory solution, mentioned earlier. From the single memory solutions we see that the application of the transforms is analgous to adding extra memory to the solution, indicating that the transforms remove the access

# Memories	Time	Cost	PI	CI
1	1987748919	73728	1.00	0.00
2	1506787767	73856	1.32	0.00
3	1025826615	72832	1.94	-0.01
4	1025826615	72832	1.94	-0.01
5	1025826615	72832	1.94	-0.01
7	1025826615	72832	1.94	-0.01
9	1025826615	72832	1.94	-0.01

Table 6.16. Timing Results for SRR Transformed Code

redundancy in the algorithm.

This represents a significant improvement on the performance as specified by the user's code, obtained through simple improvement of the user code with no additional memories added. The code semantics are the same, but the memory scheme is more efficient. Thus the transforms have had an optimising effect on the user's code. The next section details the Carried Transforms that CSiC applies to the code description.

Carried Transformations

Cluster	Dependences
2	$S_{53}\ \delta^{i}_{00-1-100}\ S_{53}$
3	$S_{50}\ \delta^{i}_{-10-1-100}\ S_{50}$
4	$S_{59}\ \delta^{i}_{-1-1-1-10}\ S_{59}$
5	$S_{15}\ \delta^{i}_{0-1-1-100}\ S_{15}$
6	$S_{66}\ \delta^{i}_{0-10}\ S_{66}$
7	$S_{45}\ \delta^{i}_{0-1-10}\ S_{45}$
8	$S_{48}\ \delta^{i}_{-1-1-1-100}\ S_{48}$

Table 6.17. Carried Dependences

The dependences found in the code that lead to carried transformations are summarised in Table 6.17. These indicate accesses whose data is carried over a number of loop iterations before it is used again. Because of this the accesses

cannot be optimised away using a single scalar variable. An on-chip shift-register has to be introduced in order to perform each optimisation. This greatly increases the efficiency of the code, at the cost of introducing more physical memory. The previous transforms, which only introduced additional scalar variables, have an effect upon the register utilisation, which may slightly increase the register count in the final design.

Dependence Cluster	Register Size	Access Saving
2	128	48093960
3	128	48093960
4	16	3757493
5	128	48093960
6	1	26460
7	1	3757320
8	128	480936832
	530	63275995

Table 6.18. Costs and Gains Associated with the Transforms

Each transform requires the addition of a shift register to the description for the array it applies to. The sizes of these registers are obtained from the iteration counts of the loops enclosing the statements to which the dependences apply and from the movement indicated by the dependence. From Section 5.3 the access reduction associated with each transform is given by,

$$\prod_{k=1}^{k=j} I_k \left(\prod_{k=i-j}^{k=i} I_k - 1 \right)$$

where j is the level of the leftmost '$<$' in the chosen dependence and i is the level from which the statement is moved, the maximum level in the dependence. Additionally, the size of the shift register introduced is

$$\prod_{k=i}^{k=j} I_k \quad (k \in S_I)$$

where j is the rightmost '0' in the dependence and i is the leftmost '0' in the

same series. S_I is the set of loop induction variables that are used to index into the array.

In the case of the dependence $S_{66}\ \delta\ ^{i}_{0-10}\ S_{66}$ the shift register has no size, ie it is a scalar variable, as the statement it refers to is

```
_tmp20=base_sig_pos_row[s]; /* 8,66,3 */
```

which is moved from level 3 to level 1, and level 3 does not not index the array.

The shift register sizes and the access gains obtained for the transforms indicated in Table 6.17 are shown in Table 6.18. The actions of the transforms are similar to those in Section 6.1.4. The case of the dependence

$$S_{53}\ \delta\ ^{i}_{00-1-100}\ S_{53}$$

is covered here by means of illustration.

```
for (;angle<=10;) /* 6,35,2 */ {
  abs_angle=angle+10; /* 4,10,2 */
  for (;start_row<=70;) /* 9,32,3 */ {
        ...
    start_col=-70; /* 8,31,3 */
    for (;start_col<=70;) /* 12,29,4 */ {
          ...
      rn=0; /* 11,27,4 */
      for (;rn<16;) /* 15,25,5 */ {
    ...
cn=0; /* 13,24,5 */
        for (;cn<8;) /* 18,22,6 */ {
      ...
  _tmp10=roffset[abs_angle][rn][cn]; /* 20,53,6 */
      ...
```

Figure 6.19. Original Source for Statement 53

The code referred to by the dependence is shown in Figure 6.19. For each iteration of the loops at level 5 and 6 there is an input redundance. This is

removed by filling a shift register at level 2 prior to the execution of the lower level loops. The statement at level 6 now takes its input from this register, so saving on external accesses and improving the memory throughput. The transformed code is shown in Figure 6.20.

```
for (;angle<=10;) /* 6,35,2 */ {
  abs_angle=angle+10; /* 4,10,2 */
  for (i1 = 0;i1 < 16;i1++) {
    for (i2 = 0;i2 < 8;i2++) {
      shift1 = roffset[abs_angle][rn][cn];
    }
  }
  for (;start_row<=70;) /* 9,32,3 */ {
        ...
    start_col=-70; /* 8,31,3 */
    for (;start_col<=70;) /* 12,29,4 */ {
          ...
      rn=0; /* 11,27,4 */
      for (;rn<16;) /* 15,25,5 */ {
    ...
cn=0; /* 13,24,5 */
        for (;cn<8;) /* 18,22,6 */ {
      ...
  _tmp10=shift1; /* 20,53,6 */
      ...
```

Figure 6.20. Transformed Source for Statement 53

6.2.5 Synthesis Summary

The timing results from applying the above transforms to the source are shown in Table 6.19. As these show there is very little improvement in performance between each solution found. The transformed code gives solutions with an improvement in performace of over 500% compared to the untransformed source code. This represents a large improvement effected by the tool. However the

access count for the algorithm is still very high and when put into the context of the application it is unacceptable, even bearing in mind CSiC 's worst case timing analysis.

# Memories	Time	Cost	PI	CI
1	481010859	73728	1.00	0.00
2	480986658	73728	1.00	0.00
3	480986649	72720	1.00	-0.01
4	480986649	72720	1.00	-0.01
5	480986649	72720	1.00	-0.01
7	480986649	72720	1.00	-0.01
9	480986649	72720	1.00	-0.01

Table 6.19. Timing Results for Fully Transformed Code

The solution obtained in [20] was to parallelise the inner loop which accesses the image array. Referring to Figure 6.12 we see that the image array access is data dependent. That is, the indices for the array are made from the other data present in the code. Dependence analysis can do nothing for us in this instance, it is necessary to rework the algorithm in order to obtain some advantage in access. In [20] this was effected by arranging the data so that contiguous blocks of image data were loaded into the ASIC and comparisons performed in parallel with the signature.

In terms of this example this amounts to an effective loop-unroll of the inner loops, allowing multiple bytes from the image array to be accessed at the same time, thereby increasing the memory bandwidth. Such transformations are feasible with the techniques used by CSiC , although more work is required to investigate algorithms to perform these operations. This has been beyond the scope of the current work.

6.3 Summary

The examples presented in this Chapter show the potential improvements offered by the CSiC tool to the synthesis task. The examples illustrated the iterative

improvement of design code through the application of high level transforms which optimise code for memory access. The power of the high level transformation was demonstrated by the comparison of the results for parallelising the architectures through the introduction of additional memory, with the results obtained for the same descriptions after the application of the transforms.

Large improvements over the un-transformed code were achieved. This shows the benefits in optimisation via these methods, namely obtaining solutions of equal or better performance at a cheaper cost. This illustrates the potency of high level synthesis and demonstrates its feasability for such design work.

Neither of the examples took more than a few minutes to process using the tool, providing the designer with good turn-around times and allowing their task to concentrate more on algorithm development than the implementation and testing details of hardware design.

A number of weaknesses in the method were uncovered. A need for statement prioritisation in the scheduling phase was identified to facilitate searching of the solution space where a number of possibilities exist for placement of the next statement to schedule.

The requirement for additional transforms to deal with loop-unrolling as a means to increase memory bandwidth was also identified. This would necessitate further development of the tool's internal models to cope with memory ports of different sizes. This would be in addition to the requirement of being able to specify different types of memory within the same design and more support for their costing.

Whilst the analysis was able to direct the transformation process towards potential savings through the use of dependence analysis of the program code, the tool's ability to assess the relative merits of design decisions was shown to be limited. This is in part due to naive costing models which are not developed to the same extent as the analysis and transformation methods. The tools ability to represent different storage types and different size vectors for memory also hampered the accurate analysis of results.

However the results show that there is a basis for synthesising memory from high level language algorithms which can optimise such descriptions for performance and memory resource. The combination of dependence-based scheduling,

allocation and transformation proves to be a powerful technique for high level synthesis of memory architectures.

Chapter 7

Summary and Conclusions

Methods for the synthesis, optimisation and evaluation of memory architectures from algorithmic descriptions have been presented. The aim of the work has been to separate the designer from the low level detail associated with this part of the design and allow them to concentrate on the development of the application algorithm. The information obtained using the described techniques can then be utilised as a basis for the synthesis of the complete design.

The segmentation of design activity between the designer and the tools is an important issue in high level synthesis. As much of the hardware-specific design knowledge as possible should reside in the tools, leaving the designer to make high level decisions which can be evaluated by simulation of the application. This aim is necessary in order to fully exploit the opportunities made available by ever increasing device densities on silicon. The pressure exerted by decreasing product development times, dictated by the increasingly widespread use of electronics and its importance in the global economy is also an important consideration.

7.1 Summary of the Thesis

In Chapter 1 the area of application was introduced and the need for a focussed synthesis approach discussed. The importance of the memory architecture in image processing applications where the access overhead is likely to be the limiting factor in system performance, leads to this research.

The main synthesis systems that address this area are reviewed in Chapter 2. Much of the memory synthesis in these systems is performed with guidance from the user. In the CATHEDRAL tools [34], for example, this is achieved by the designer indicating the areas of the input description that will benefit from memory optimisation.

The chapter also reviewed techniques developed for super-optimising compilers which perform dependence analysis for array statements in high level code. These methods form the basis of a number of transformations used in parallelising program code for multi-processor machines. The work presented in this thesis employs such techniques for solving some of the problems of memory synthesis for high throughput applications.

In Chapter 3 the development of a tool for extracting information necessary for the synthesis task was described. This tool, CSiÇimplements various data flow techniques for profiling image processing applications written in the C programming language. These methods are invaluable in reducing an algorithmic description expressed in software into a format suitable for hardware synthesis. The information so obtained is then used for dependence testing within the algorithm. This provides a profile of the algorithm's interaction with memory, and indicates regions of code that can be optimised so as to improve the memory performance.

Chapter 4 describes the synthesis of memory architectures from this information. An interleaved scheduling and allocation method is presented which performs transformational synthesis of different architectures for the support of the input algorithm. This method generates memory organisations and complementary schedules, with data grouped in memory according to its relative importance with respect to access. This grouping is effected using both global and local considerations, based on the access profile obtained in previous stages of the synthesis.

The scheduling method derives from techniques reviewed in Chapter 2. It is list-based in nature, utilising array dependence information for prioritising the order of array access in basic blocks. Conflicts in the scheduling are solved using additional information obtained from the input description concerning the processing of the array data.

The generation of multiple-memory architectures allows the designer to evaluate a range of structures suitable for supporting the input algorithm. The CSiC tool ensures that these architectures correctly support the application and provides an optimal design for each choice in terms of memory performance.

The synthesised structures are evaluated using the generated schedules in conjunction with loop iteration and block ordering information obtained from the input description. This gives a timing profile for each architecture, showing their relative performances. Costing of the architectures is based on the sizes of memory required to hold the separate arrays of the input description.

In order to improve the performance of the generated hardware, the tool implements a number of high level transformations that can be applied to the input description directly. The transforms themselves utilise the inter-statement dependence information obtained in previous stages in order to detect regions of the algorithm that can be improved.

Chapter 5 covers those transforms currently utilised in CSiC . These aim to reduce the access bottlenecks of the algorithm by altering its access behaviour, whilst keeping semantic parity with the original input. The primary action of the transforms is to remove unnecessary accesses in the algorithmic description by either moving the position of statements or altering the location of the data being accessed.

The first type of transforms, the Simple Transforms, amount to a normalisation of the input description's coding. It transliterates the use of data by statements into a form more suitable for translation into an optimal memory organisation. This is possible because of the tool's assumption that array statements in the input description refer to memory accesses and that other variables are held as values in individual registers, apart from the main memory. This frees the designer from the task of having to specify explicit locations for the data in the code, increasing the "high-levelness" of the synthesis process.

The second type of transform, The Carried Transforms, alter the underlying memory architecture expressed by the code. This is achieved by assigning different types of storage to elements of the arrays during loop lifetimes. Data that is required a multiple number of times during a loop lifetime is duplicated in

the local storage of the hardware scheme, reducing the access overhead by minimising the amount of communication between the processing core and the main memory or memories. Coherence between the two data is maintained within the description.

Chapter 6 presents results for the CSiC tool using two example programs as input. These are taken from real applications and illustrate the efficacy of the techniques introduced in previous chapters. The two major methods of optimisation presented in this thesis, namely memory splitting and source-level transformation, are discussed for each example.

7.2 Discussion of Results

From the examples presented in Chapter 6 a number of conclusions can be drawn. The most important of these is that the methods presented provide a means of optimising algorithmic descriptions in the application domain for memory access. The combination of high level transformation on the input description and the scheduling and allocation of the memory operations provides the user with a fast and effective means of exploring the available solution space.

The scheduling and allocation task performed by the CSiC tool ensures that the allocation of data to memory blocks provides an optimal arrangement that is able to support maximum parallelism in the schedules with the available resources. A number of different solutions are produced by the tool, allowing the user to make their own trade-offs based on cost and performance.

Memory fragmentation is minimised via the "packing" of smaller, less critical arrays into the allocated memories. This leads to minimal memory sizes whilst maintaining an optimal arrangement of data in memory with respect to the schedules generated. Memory sizes tend towards factors of two. This strategy will produce cost effective solutions where applicable by allowing the construction of memories from standard parts.

The cost estimations used by the tool for relative comparisons are crude and take no account of issues such as address generation nor on-chip costs for extra registers. However, the designer can get a feel for the relative merits of different

architectural choices and their costs. Iterative improvement of the user description is facilitated with the application of high level transforms to the original input. This was shown to produce significant improvements in performance for increases in memory costs that ranged between 1 and 20%.

The transformations applied to the source code were shown to generate solutions better or equal to the solutions obtained by parallelisation through addition of memories. This method increases performance at minimum cost and tends towards optimum results for the resources available to the algorithm. Transforms can be applied iteratively and hence can guide the synthesis process toward an optimal solution.

7.3 Future Work

A number of issues have arisen from this work which require some further exploration. These fall mainly into four categories: those concerning scheduling of memory access; further development of the tool to cope with differing memory types in the internal representation; additional exploration in the use of high level transforms and their effect upon the synthesised architectures; and finally the investigation into the interaction of memory synthesis with functional synthesis.

Scheduling

The memory scheduling performs well; beginning from an optimal schedule, in terms of precedence, it ensures that the resources available for each architecture are fully utilised. The technique presented in this thesis could be extended to make use of head and tail overlap in the loop schedules. This would be akin to the loop pipelining described in [7] and utilised in percolation based synthesis [61]. This could improve the scheduling performance for multiple memory solutions, utilising idle cycles in parallel memories at the beginning and end of each loop; a situation that can occur with the current methodology.

To a certain extent the application of the Redundancy Removal Transforms[1],

[1] see Section 5.1.3

particularly the CRRT, implicitly pipelines the loops by taking accesses out of the loop schedule and supplying the data concurrently with that referenced from memory. However, this will not remove all idle cycles from a schedule and would be greatly complemented by pipelining.

The consideration of conditional constructs and their impact on scheduling would need some investigation with respect to pipelining. As was shown, the Access Motion Transform can be used to remove common accesses from separate conditional branches which improves individual block schedules. But in cases where dissimilar access profiles exist between conditional branches the ramifications for pipelining would require further investigation.

Internal Representation

The memory types that CSiC currently supports are limited to internal registers and memory blocks composed of standard RAM. The RR Transforms should result in the creation of shift registers and other intermediate memory types for local storage. This would increase the precision of the costing procedure and lead to a more complete specification of memory hierarchy. Such storage could then be shared between exclusive processes, leading to a more optimal solution in terms of on-chip resources.

Additional support for multi-port memories, which are increasingly used in digital design, would further increase the power of the tool. This would have an effect on the scheduling and allocation algorithms which would require changing to encompass the new memory types. Such changes would not be major, the existing scheme for building allocation matrices for scheduling and allocation could be easily updated to take advantage of the increase in throughput potential.

Additional High Level Transforms

As the fingerprint verification example showed, there is scope for further transforms to be implemented by CSiC . The implementation of this application achieves a large data throughput by parallelising the inner loop of the comparison algorithm. In this way a number of blocks within a signature print are accessed and compared simualtaneously, dramatically increasing the performance of the

algorithm.

This could be achieved within CSiC by one of two methods. Successively unrolling the loop body and coalescing groups of contiguous access statements is one transform that could be applied. The same results could be obtained through dependence analysis of the loops' access statements. Such methods are used in superoptimising compilers for parallelising inner loops. Both would lead to wider bus widths between core and memory, altering other aspects of the design, and requiring further extension of the tool for representing data sizes.

Other transforms that would prove useful involve the merging and splitting of different "process" loops within a description. CSiC already effectively achieves the latter via the scheduling of processes prior to block scheduling. This parallelisation is akin to loop merging, although there is no sharing of control between the parallelised processes. Loop splitting would be a useful technique for altering the behaviour of processes prior to the application of further transforms.

Interaction with Functional Synthesis

In order for CSiC to complete the synthesis loop it would have to be linked to the synthesis of data path and control logic. The influences between these different types of synthesis are still an open topic for research, and a very necessary one if designers are going to benefit from effective synthesis in high throughput domains.

Whilst the required data for the synthesis of the memory interface is contained in the access profile of the algorithm, feedback from functional analysis could prove useful in guiding memory synthesis. Computationally non-critical access could be flagged with timing information. Such feedback could be useful in refining the optimisation applied by the memory synthesis stage.

Various trade-offs are made during memory synthesis, for instance widening a data bus verses splitting data between two memories. Whilst both solutions might obtain the same speedup for memory, they could have quite different ramifications for processing core. Again, feedback from functional synthesis would have a role to play in such situations. Similar examples would be trade-offs between placing on-chip data into individual registers or placing it in some other on-chip memory structure like a RAM or shift register. Again, this would have an effect upon

functional synthesis and feedback could prove useful in making the choice.

7.4 Closing Comments

The area of high level synthesis is still developing and has, in many ways, still to reach many of its stated aims. Whether these are ever achieved or not is only one issue, one dependent as much upon the movements of the marketplace as upon further research. Commercial designers still largely use schematic capture tools and many resist conversion to script-based design techniques which require the utilisation of languages such as Verilog and VHDL.This is in part due to the quality of the tools available and their likelihood, or its perception, of their completing better designs than the designers themselves. Such resistance is gradually being eroded as more vendors add synthesis packages to their tools.

However as densities inexorably increase and the design time windows continue to contract, utilising every available transistor becomes less important than obtaining a provable design on time. Increasingly these goals will only be achievable using synthesis from higher levels of abstraction. This is an issue that is most pressing to the electronics industry as a whole and one which demands solutions if the technologies of the next millenium are to be fully utilised.

References

[1] Robert X Cringely. *Accidental Empires.* Penguin Books, 1993.

[2] J Allen. Performance Driven Synthesis. *Proceedings of The IEEE*, February 1990.

[3] T D Friedman and S C Yang. Methods Used in an Automatic Logic Design Generator (ALERT). *IEEE Transactions on Computing*, 18:593–614, 1969.

[4] S W Director, A C Parker, D P Siewiorek, and D E Thomas. A Design Methodology and Computer Aids for Digital VLSI Systems. *IEEE Transactions on Computing*, CAS-28(7):634–645, July 1981.

[5] T J Kowalski. *An Artificial Intelligence Approach to VLSI Design.* Kluwer Academic Publishers, 1985.

[6] H W Trickey. *Compiling Pascal Programs into Silicon.* PhD thesis, Stanford University, July 1985.

[7] A Aiken and A Nicolau. Perfect Pipelining: A New Loop Parallelisation Technique. In *Proceedings of the European Symposium on Programming*, number 300, pages 221–235. Springer-Verlag Lecture Notes in Computer Science, March 1988.

[8] W.Sakowski, M.Ossysek, and B.Nowak. VHDL as a Specification Language for a High-level Synthesis System. In *International Conference Of Microelectronics : Microelectronics 92*, pages 48–57, 1992.

[9] T A Ly and J T Mowchenko. Applying Simulated Evolution to High Level Synthesis. *IEEE Transactions on CAD*, 12(3), March 1993.

[10] A J Martin. Synthesis of Asynchronous VLSI Circuits. Tutorial at VLSI 91 Conference, Edinburgh, August 1991.

[11] B S Haroun and M I Elmasry. SPAID: An Architectural Synthesis Tool for DSP Custom Applications. *IEEE Transactions on CAD*, 8(4):431–447, April 1989.

[12] E D Lagnese and D E Thomas. Architectural Partitioning for System Level Synthesis of Integrated Circuits. *IEEE Transactions on CAD*, 10(7):847–860, 1991.

[13] R A Walker and D E Thomas. Behavioural Transformation for Algorithmic Level IC Design. *IEEE Transactions on CAD*, 8(10):1115–1128, October 1989.

[14] N C Park and A C Parker. Sehwa: A Software Package for the Synthesis of Pipelines From Behavioural Specifications. *IEEE Transactions on CAD*, pages 356–370, March 1988.

[15] M Potkonjak and J Rabaey. Retiming for Scheduling. In *IEEE Workshop on VLSI Signal Processing IV*, pages 23–32, November 1990.

[16] P G Paulin and J P Knight. Force Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on CAD*, June 1989.

[17] C Y Wang and K K Parhi. Dedicated DSP Architecture Synthesis Using the MARS Design System. In *Proceedings of the International Conference On Acoustics, Speech, And Signal Processing*, pages 1253–1256, Toronto, May 1991.

[18] I Verbauwhede, F Catthoor, J Vandewalle, and H de Man. Background Memory Synthesis for Algebraic Algorithms On Multi-processor DSP Chips. In *Proceedings of VLSI*, pages 209–220, Munich, August 1989.

[19] P B Denyer, D Renshaw, G Wang, M Lu, and S Anderson. On-chip CMOS Sensors for VLSI Imaging Systems. In *Proceedings of VLSI 91*, volume 1, pages 157–166, Edinburgh, 1991.

[20] S Anderson. *A VLSI Smart Sensor-Processor for Fingerprint Comparison.* PhD thesis, University of Edinburgh, 1993.

[21] W Moore and W Luk. *FPGAs.* Oxford University Press, 1991.

[22] C M Chu, M Potkonjak, M Thaler, and J Rabaey. Hyper - an Interactive Synthesis Environment for High-performance Real-time Applications. In *Proceedings - IEEE International Conference On Computer Design : VLSI In Computers & Processors*, pages 432–435, 1989.

[23] D Verkest, P Johannes, L Claesen, and H de Man. Program Transformation of Hardware Descriptions By Means of ILP. In *1989 IEEE International Symposium On Circuits And Systems*, pages 1174–1177, 1989.

[24] I Vandeweerd, K Croes, L Rijnders, P.Six, and H.de Man. Redusa - Module Generation By Automatic Elimination of Superfluous Blocks in Regular Structures. In *26th ACM/IEEE Design Automation Conference*, pages 694–697, 1989.

[25] C Ramsay, K Sutherland, D Renshaw, and P B Denyer. A Comparison of Vector Quantization Codebook Generation Techniques Applied to Automatic Facial Recognition. In *Proceedings of the British Machine Vision Conference*, September 1992.

[26] J Linde, A Buzo, and R M Gray. An Algorithm for Vector Quantizer Design. *IEEE Transactions on Communications*, 28(1):84–95, January 1980.

[27] W H Bruce. *Fingerprint Comparison By Template Matching.* PhD thesis, University of Edinburgh, 1993.

[28] M C McFarland, A C Parker, and R Camposano. The High Level Synthesis of Digital Systems. *Proceedings of the IEEE*, 78(2), February 1990.

[29] D E Thomas, J K Adams, and H Schmit. A Model and Methodology for Hardware-software Codesign. *IEEE Design and Test of Computers*, 10(3):6–15, 1993.

[30] D D Gajski and R Khun. Guest Editor's Introduction: New VLSI Tools. *IEEE Computer*, 16(12):11–14, December 1983.

[31] E D Lagnese and D E Thomas. Architectural Partitioning for System Level Design. In *26th ACM/IEEE Design Automation Conference*, pages 62–67, June 1989.

[32] A Chatterjee and R Hartley. A New Simultaneous Circuit Partitioning and Chip Placement Approach Based On Simulated Annealing. In *27th ACM/IEEE Design Automation Conference*, pages 36–39, 1990.

[33] S Kirkpatrick, C Gelatt, and M Vecchi. Optimisation By Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[34] F Catthoor, J Rabaey, and H de Man. Target Architectures in the CATHEDRAL Synthesis Systems - Objectives and Impact. In *Proceedings of the IEEE International Symposium On Circuits And Systems*, pages 1907–1910, 1989.

[35] D Genin, P Hilfinger, J Rabaey, C Scheers, and H de Man. DSP Specification Using the Silage Language. In *Proceedings of the International Conference On Acoustics, Speech, And Signal Processing*, pages 1057–1060, 1990.

[36] H De Man, J Rabaey, P Six, and L Claesen. CATHEDRAL II: A Silicon Compiler for Digital Signal Processing. *IEEE Design & Test Of Computers*, 3(6):13–25, 1986.

[37] S Note, W Geurts, F Catthoor, and H De Man. CATHEDRAL III - Architecture Driven High Level Synthesis for High Throughput DSP Applications. In *28th ACM/IEEE Design Automation Conference*, pages 597–602, 1991.

[38] F Catthoor, M Vanswaaij, J Rosseel, and H De Man. Array Design Methodologies for Real Time Signal Processing in the CATHEDRAL IV Synthesis Environment. In *Algorithms And Parallel VLSI Architecture II*, pages 211–221, 1992.

[39] L Laesen, F Catthoor, D Lanneer, G Goossens, S Note, J Vanmeerbergen, and H de Man. Automatic Synthesis of Signal-processing Benchmark Using the CATHEDRAL Silicon Compilers. In *Proceedings Of The IEEE Custom Integrated Circuits Conference*, pages 381–384, 1988.

[40] C M Chu, M Potkonjak, M Thaler, and J Rabaey. Hyper - an Interactive Synthesis Environment for High-performance Real-time Applications. In *Proceedings - IEEE International Conference On Computer Design : VLSI In Computers & Processors*, pages 432–435, 1989.

[41] M Bayoumi, N Ramakrishna, N Madraswala, and S Sahu. Sphinx - a High Level Synthesis System for DSP Design. In *Proceedings of the IEEE International Symposium On Circuits And Systems*, pages 172–175, 1992.

[42] R Camposano. Structural Synthesis in the Yorktown Silicon Compiler. In *VLSI '87*, pages 61–72, 1987.

[43] E F Girczyc, R J A Buhr, and J P Knight. Applicability of a Subset of ADA as an Algorithmic Hardware Description Language for Graph Based Hardware Compilation. *IEEE Transactions on CAD*, CAD-4(2):134–142, April 1985.

[44] V Berstis. The V Compiler: Automating Hardware Design. *IEEE Design and Test of Computers*, 6(2):8–17, April 1989.

[45] T Tanaka, T Kobayashi, and O Karatsu. HARP: Fortran to Silicon. *IEEE Transactions on CAD*, 8(6), June 1989.

[46] G DeMicheli, D Ku, F Mailhot, and T Truong. The Olympus Synthesis System. *IEEE Design and Test of Computers*, 7(5):37–53, October 1990.

[47] L E Thon and R W Broderson. C-to-silicon Compilation. In *Proceedings Of The IEEE Custom Integrated Circuits Conference*, 1992.

[48] R Broderson. *Anatomy of a Silicon Compiler*. Kluwer Academic Publishers, 1992.

[49] D E Thomas and P Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[50] J Bhasker and H C Lee. An Optimiser for Hardware Synthesis. *IEEE Design and Test of Computers*, 7(5):20–36, October 1990.

[51] M S Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[52] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[53] D Landskov, S Davidson, B Shriver, and P Mallet. Local Microcode Compaction Techniques. *ACM Computing Surveys*, 12(6), September 1980.

[54] A C Parker. MAHA: a Program for Datapath Synthesis. In *23rd ACM/IEEE Design Automation Conference*, pages 461–466, June 1986.

[55] B M Pangrle and D D Gajski. SLICER: a State Synthesiser for Intelligent Silicon Compilation. In *Proceedings IEEE Conference on Computer Design*, October 1987.

[56] M C McFarland. BUD: Bottom-up Design of Digital Systems. In *23rd ACM/IEEE Design Automation Conference*, pages 474–479, July 1986.

[57] M Potkonjak and J Rabaey. A Scheduling and Resource-allocation Algorithm for Hierarchical Signal Flow-graphs. In *26th ACM/IEEE Design Automation Conference*, pages 7–12, 1989.

[58] P G Paulin and J P Knight. Force-directed Scheduling in Automatic Data Path Synthesis. In *24th ACM/IEEE Design Automation Conference*, pages 195–202, July 1987.

[59] P G Paulin and J P Knight. Scheduling and Binding Algorithms for High Level Synthesis. In *26th ACM/IEEE Design Automation Conference*, pages 1–6, 1989.

[60] J Scheichenzuber, W Grass, U Lauther, and S Marz. Global Hardware Synthesis From Behavioural Dataflow Descriptions. In *27th ACM/IEEE Design Automation Conference*, pages 456–461, 1990.

[61] R Potasman, J Lis, A Nicolau, and D Gajski. Percolation Based Synthesis. In *27th ACM/IEEE Design Automation Conference*, pages 444–449, 1990.

[62] G Goossens, J.Vandewalle, and H.de Man. Loop Optimization in Register-transfer Scheduling for DSP-systems. In *26th ACM/IEEE Design Automation Conference 89*, pages 826–831, 1989.

[63] D E Thomas, C Y Hitchcock, T J Kowalski, J V Ragan, and R A Walker. Automatic Datapath Synthesis. *IEEE Transactions on Computing*, 16(12):59–70, December 1983.

[64] E F Girczyc. Loop Winding - a Data Flow Approach to Functional Pipelining. In *Proceedings of the IEEE International Symposium On Circuits And Systems*, volume 3, pages 382–385, 1987.

[65] H W Trickey. Flamel: A High Level Hardware Compiler. *IEEE Transactions on CAD*, CAD-6(2):259–269, March 1987.

[66] C Tseng, S G Rothweiler, S Sutarwala, and A M Prabhu. MIND - a Module Binder for High Level Synthesis. In *International Conference on Circuits and Devices*, pages 420–423, 1989.

[67] C A Papachristou and H Konuk. A Linear Program Driven Scheduling and Allocation Method Followed By an Interconnect Algorithm. In *27th ACM/IEEE Design Automation Conference*, pages 77–83, June 1990.

[68] R Camposano and J T V VanEijndhoven. Partitioning a Design in Structural Synthesis. In *International Conference on Computer Aided Design*, pages 564–566, November 1987.

[69] R A Bergamaschi, D.Lobo, and A.Kuehlmann. Control Optimization in High-level Synthesis Using Behavioral Don't Cares. In *29th ACM/IEEE Design Automation Conference*, pages 657–661, 1992.

[70] W Geurts and F Catthoor. DSP Applications Suited for Lowly Multiplexed Architectures. Technical report, IMEC, VSDM division, August 1990.

[71] D D Gajski, editor. *Silicon Compilation.* Addison-Wesley, 1988.

[72] CY Lee, F Catthoor, and H De Man. Breaking the Bottleneck of Sequential Decoding for High Speed Digital Communication. In *Proceedings of the International Conference On Acoustics, Speech, And Signal Processing*, volume 1, pages 1213–1216, 1991.

[73] I Verbauwhede, F Catthoor, J Vandewalle, and H de Man. In-place Memory Management of Algebraic Algorithms On Application Specific ICs. *Journal of VLSI Signal Processing*, 3(3):193–200, September 1991.

[74] P Lippens *et al.* PHIDEO: A Silicon Compiler for High Speed Algorithms. In *Proceedings of the European Conference on Design Automation*, pages 436–441, Amsterdam (Netherlands), February 1991.

[75] U Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[76] M J Wolfe. *Optimising Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[77] J R Allen. *Dependence Analysis for Subscripted Variables and It's Application to Program Transformations.* PhD thesis, Rice University, 1983.

[78] U Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[79] U Banerjee. Data Dependence in Ordinary Programs. Master's thesis, University of Illinois at Urbana-Champaign, 1976.

[80] J R Allen and K Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[81] E A Snow. *Automation of Module Set Independent Register-Transfer Level Design.* PhD thesis, Carnegie Mellon University, April 1978.

[82] B Brode. Precompilation of Fortran Programs to Facilitate Array Processing. *Computer*, 14(9):46–51, September 1981.

[83] J R Beckman Davies. Parallel Loop Constructs for Multiprocessors. Master's thesis, University of Illinois at Urbana-Champaign, May 1981.

[84] C D Polycronopoulos. Advanced Loop Optimisations for Parallel Computers. In *1st International Conference on Super-Computing*, pages 255–277, 1987.

[85] D Callahan, S Carr, and K Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceeding of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990.

[86] S B Lippman. *C++ Primer.* Addison Wesley, 1991.

[87] R M Stallman. *The GNU C Compiler.* Free Software Foundation, 675 Mass Ave, Cambridge MA 02139, USA, 1993.

[88] S Naeher. *LEDA - A Library of Efficient Data Types and Algorithms.* Max-Planck-Institut fuer Informatik, 1993.

[89] M J Wolf. *Tiny.*

[90] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[91] William Pugh and Dave Wonnacott. Eliminating False Data Dependences Using the Omega Test. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1992.

[92] A S Tannenbaum, H van Staversen, E G Keizer, and J W Stevenson. A Practical Tool Kit for Making Portable Compilers. *Communications of the ACM*, 26(9), September 1983.

[93] C Fraser and D Hanson. *LCC - A Lightweight C Compiler.* AT&T Bell Laboratories.

[94] P B Denyer. The Alice Project. Technical report, VLSI Vision Ltd, 1992.

[95] P B Denyer. The Door Project. Technical report, VLSI Vision Ltd, 1992.

[96] E R Davies. *Machine Vision: Theory, Algorithms, Practicalities.* Academic Press Limited, London, San Diego, 1990.

[97] J W Davidson and A M Holler. A Study of a C Function Inliner. *Software Practice and Experience*, 18(8):775–790, 1988.

[98] R Corbett and R M Stallman. *The GNU Compiler Generator.* Free Software Foundation, 675 Mass Ave, Cambridge MA 02139, USA, 1993.

[99] V Paxson and J Poskanzer. *FLEX - Fast Lexical Analyser Generator.* Computer Science Dept, Cornel University, Ithaca, NY 14853-7501, 1990.

[100] J Lee. *ANSI C Grammar.* Gatech.

[101] John Cocke. Global Common Subexpression Elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, July 1970.

[102] B R Rau. Data Flow and Dependence Analysis for Instruction Level Parallelism. In *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 218–235, August 1991.

[103] A Nicolau, R Potasman, and H Wang. Register Allocation, Renaming and Their Impact On Fine-grain Parallelism. In *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 218–235, August 1991.

[104] C T Hwang, J H Lee, Y C Hsu, and Y L Lin. A Formal Approach to the Scheduling Problem in High Level Synthesis. *IEEE Transactions on CAD*, 10:464–475, April 1991.

[105] S Devadas and A R Newton. Algorithms for Hardware Allocation in Data Path Synthesis. *IEEE Transactions on CAD*, 8(7), July 1989.

[106] P Paulin. *High Level Synthesis Using Global Scheduling and Binding Algorithms.* PhD thesis, Carlton University, Ottawa, February 1988.

[107] D Grant, I Finlay, and P B Denyer. Synthesis of Address Generators. In *International Conference on Computer Aided Design*, pages 116–119, 1989.

[108] D J Kuck, R H Kuhn, D A Padua, B Leasure, and M Wolfe. Dependence Graphs and Compiler Optimisations. In *8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[109] M J Wolfe. Advanced Loop Interchanging. In *Proceedings of the International Conference on Parallel Processsing*, pages 536–543, 1986.

[110] M J Wolfe. Iteration Space Tiling for Memory Hierarchies. In *SIAM Conference on Parallel Processing*, pages 357–361, December 1987.

[111] C Tseng and D P Siewiorek. Automated Synthesis of Data Paths in Digital Systems. *IEEE Transactions on CAD*, 5(3):379–395, July 1986.

[112] K G N Sutherland. *Automatic Facial Recognition Based On Facial Feature Analysis.* PhD thesis, University of Edinburgh, 1992.

[113] C S Ramsay, K Sutherand, D Renshaw, and P B Denyer. A Comparison of Vector Quantization Codebook Generation Techniques Applied to Automatic Facial Recognition. In *Proceedings of the British Machine Vision Conference*, Leeds, September 1992.